PRIME: A BOTTOM-UP APPROACH TO PROBABILISTIC RULE DEVELOPMENT

NAGW-1333

by

Scott A. Miller

Center for Intelligent Robotic Systems
for Space Exploration
Electrical, Computer and Systems Engineering
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

CIRSSE Report #55

Contents

List of Figures

List of Tables

Acknowledgements

The author wishes to express his gratitude to Prof. George N. Saridis, for his support and encouragement. Funding provided by NASA and E. I. Du Pont de Nemours & Co. during work on this thesis is gratefully acknowledged. Special thanks are extended to Michael C. Moed for his invaluable assistance in all phases of the project, and to Stephen Blaes for proofreading this document.

Abstract

PRIME is a system to be used by an intelligent machine to allow it to operate in an abstract but uncertain (or stochastic) environment. It maintains a model of the effects of the machine's actions in the form of a rule base, which is induced from experience. This bottom-up approach to rule development allows the model to adapt to changes in the environment.

Each rule consists of a condition under which the rule is active, an action, the effect of the action on the environment, and an estimate of the probability of this effect occurring. The effect probabilities are used to model the uncertainty in the environment, permitting multiple possible effects for a single action under a particular set of conditions.

The objective of the intelligent machine is to satisfy user-specified goals with maximum probability of success. PRIME fulfills this requirement in two ways: it continuously updates the rule base with the most recent information, to ensure the validity of the model; and it generates plans which have the maximum probability of achieving the goals, based on the probability estimates in the rule base.

PRIME is composed of three main processes: exploration, generalization and planning. In exploration, the machine executes various randomly chosen actions, observes the effects on the environment, and updates the rule base accordingly. This process is used to develop the rule base in simulation, as well as to supplement the current knowledge during normal operation. Generalization is the procedure used to induce general rules from experience, which is encoded in the form of specific rules. These general rules extend the machine's knowledge to situations which have not been encountered yet, thereby increasing the capability of the machine to plan effectively. Planning is the process of constructing an optimal sequence of actions to satisfy a goal, using the rule base to predict

the effects of these actions and to determine the probability of success of the plan. The rule representation and many other data structures were specifically chosen to maximize the efficiency of these processes.

A simulated environment was designed to test the performance of PRIME. The results of experimentation were largely negative. The main problem was that the domain coverage of the rules was inadequate for the number of rules stored in the rule base, due to redundancies in general rules and numerous rules covering ineffective actions. It was determined that a more efficient generalization, and some form of goal-directed exploration, are necessary in order to solve most of the current deficiencies in PRIME.

1. Introduction

1.1. Function of PRIME

The need for autonomy in intelligent machines is becoming more apparent as the tasks we expect these machines to perform grow in complexity. In unmanned space explorations, real-time remote feedback becomes difficult if not unfeasible. In dangerous environments such as nuclear power plants, autonomous intelligent problem solving may become necessary in order to swiftly attend to an emergency situation. In very complex projects, such as building the Space Station, the use of a team of autonomous intelligent machines allows humans to spend their time planning and monitoring the high-level, mission oriented tasks.

For the purposes of this thesis, the primary purpose of an autonomous intelligent machine is to satisfy user-specified goals with maximum probability of success. This is achieved by developing plans using the knowledge from a continually updated internal model of the machine's effects on its surroundings. This model is maintained in the face of a changing environment, incomplete knowledge of surroundings, and new situations not previously encountered.

PRIME (Probabilistic Rule Induction MEchanism) is a subsystem which can be used by an autonomous intelligent machine to guide the operation of a system of robots, and provide the internal model building function mentioned above. PRIME interacts with its environment at a high level, i.e., it receives abstract information about the objects in its environment, and issues high-level commands to manipulate those objects.

PRIME can function as part of the Organization Level of Saridis' intelligent machine architecture called the Hierarchical Intelligent Control System [32, 33]. This architecture

provides a hierarchical breakdown of tasks by a modular, three-tier structure (see Figure 1.1), which is organized according to the principle of Increasing Precision with Decreasing Intelligence. The role of each level in the architecture is described briefly below:

Organization Level: performs high-level reasoning and planning; issues plans consisting of sequences of high-level actions to Coordination Level.

Coordination Level: decomposes high-level actions from Organization Level into sequences of low-level tasks for each hardware controller in the Execution Level.

Execution Level: performs the actions specified by the Coordination Level in the environment; these actions are implemented as control functions with analytical performance measures.

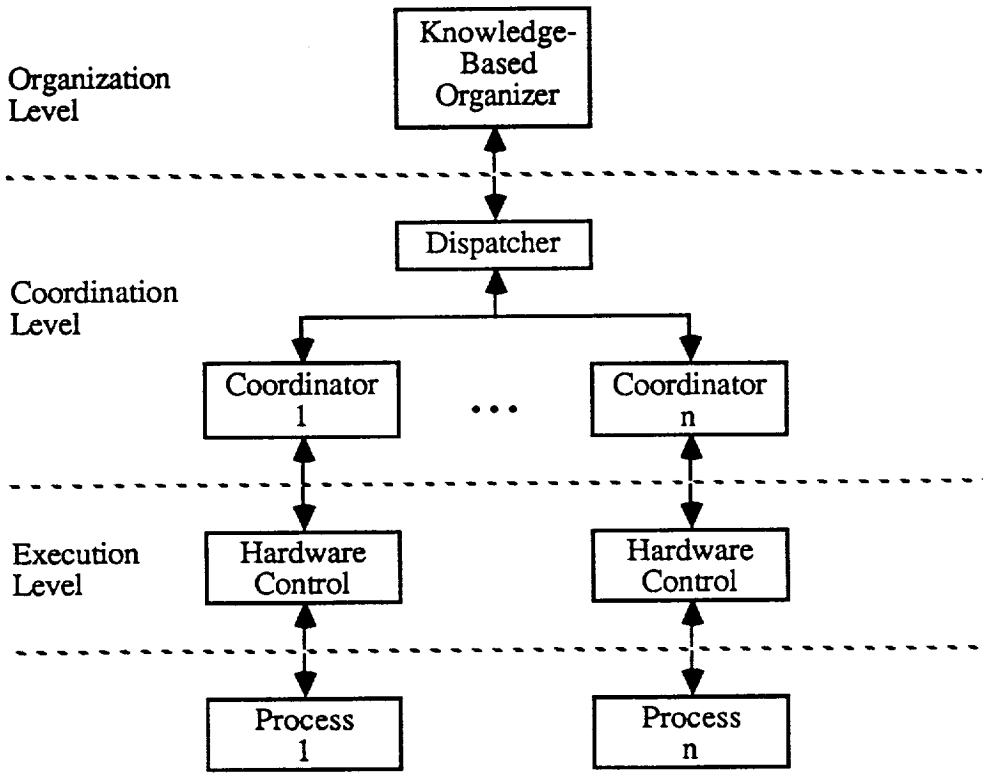

Figure 1.1 Hierarchical Intelligent Control System

Each level has its own performance measure, specified as an entropy value, which it attempts to minimize during its operation. Each level provides its entropy as feedback to the level above it, which is used to modify the operation of the higher levels. The goal of the intelligent machine is to minimize its total entropy. Thus, the operation of the machine is treated as an optimization problem—entropy minimization provides an analytical basis for the design of intelligent machines.

PRIME's functions place it naturally in the Organization Level. The inclusion of PRIME within the Organization Level is discussed in §3.6, where it is demonstrated that PRIME's goal of maximizing the probability of success of a plan is equivalent to minimizing its entropy. PRIME can also work in conjunction with Moed's Boltzmann machine architectures [21, 22, 23] which operate in the Organization Level; the details of this interfacing are considered in §6.1.

1.2. Environment constraints

This section defines the properties of the environment in which PRIME is designed to operate.

As mentioned in the last section, PRIME observes abstract states of the environment. These states are high-level properties of known objects, without detailed information which is more appropriate to the Coordination Level. Examples of such states are:

MSC1 near IEA pallet 2

FTS1 attached to SSRMS

RSM stowed on MSC2

(Definitions of these abbreviatiosn appear in Appendix D.)

Environmental changes are caused solely by the machine's actions. The effect of an action is seen immediately after executing that action, and these effects cannot be delayed or extended over time. This implies that the environment is not dynamic, in the sense that no change in state occurs between actions other than the immediate one dictated by the last action. Within a structured or sufficiently abstracted environment, this is not a very restrictive assumption.

The only *a priori* knowledge about the environment consists of a fixed set of objects, a fixed set of object classes, and a fixed set of states concerning these objects. The object classes are sets of objects, whose members are defined beforehand. The states concerning each object are also defined beforehand; these are the abstract states discussed above, which take on the values *true* or *false*.

Finally, it is assumed that not all relevant environmental states may be observed. That is, the machine may have to reason with incomplete information. While this is a very realistic assumption, it causes problems in many of the traditional knowledge-based systems, as discussed in §2.1.

1.3. Required features of PRIME

Given the information in the previous two sections, there are certain features which appear to be required in the design of PRIME. These features are described in this section.

1.3.1. Rule structure

The internal model is in the form of a rule base. The rule structure dictated by functional specifications consists of four fields:

condition: the required values of environmental states in order to activate the rule.

action: the high-level action with which this rule is concerned.

effect: the resulting environmental state values after the action above is performed.

probability: an estimate of the probability that the specified action will actually result in the specified effect.

This rule structure can be viewed as an augmented production rule. A production rule dictates an action when a set of conditions are met; thus, it has the first two of the components above. However, by adding the effect field, the purpose of the rule changes from dictating an action to predicting the effect of an action under certain conditions. This is exactly the type of predictive information needed for planning, which is a necessary component of PRIME. Without planning, the rules would have to be goal-dependent, and it would be difficult to use the machine's knowledge when new goals are specified.

The probability field of the rule is included because an action might appear to have more than one effect in a particular situation. This is because the machine might not have access to all relevant states, so two situations which appear identical might actually be different, leading to different effects when the same action is executed. This uncertainty, along with any unreliability in feature extraction or action execution, is irreducible, so the best the machine can do is maintain a measure of this uncertainty, in the form of an objective probability of the effect occurring. Using these effect probabilities, the environment is modelled as a Markov process: for any given situation and action, a set of effects (or next states) and a probability distribution over those effects is stored in the rule base. In fact, the environment might be a nonstationary Markov process, because of changes in the environment over time, such as the decaying reliability of a tool with extensive use. Also, the machine might move from one environment to a similar environment with slightly different probability distributions, such as the move from a

simulation to the real environment. Therefore, the probability estimators should be able to track continuous or step changes in effect probabilities.

The probability value in each rule provides a measure of how certain the effect of a given action is in a particular situation. These values can be used to decide between alternative methods of achieving a goal, so that actions with higher probabilities of achieving desired effects are preferred over ones with lower probabilities. Thus, a planner can be designed which searches the space of action sequences using the rules and their associated probabilities, to find the plan with the maximum probability of success.

1.3.2. Induction of specific rules

An additional requirement is the ability to induce specific rules from experience. *Induction* is used in a general sense in this document as "the development of additional internal knowledge based on specific data." A *specific rule* contains specific condition, action and effect fields; it predicts the effect of performing a particular action under one and only one environmental state. In the context of specific rules, induction means the storing of experience in a form which can be used later; it also refers to the learning of the probability of effect associated with each specific rule.

It may not be necessary to start with an empty rule base, but the machine should be able to modify or augment its rule base in light of new experiences, by adding or updating specific rules which summarize these experiences. There are several reasons for this requirement:

1. Induction eases knowledge transfer from human to machine, by removing the burden of knowledge engineers to construct a consistent and complete set of rules for a particular domain. This is a well-known problem with developing expert systems, and Michalski [20] has suggested that inductive learning can be used for both the initial

development and the ongoing maintenance of an expert system rule base. The same reasoning can be applied here.

2. Since the environment is assumed to be nonstationary, the rule base must be free of immutable preconceptions about the machine's effect on its surroundings. Induction provides a mechanism for adjusting the internal model of the environment to reflect the most recent information.
3. It is unreasonable to expect a human or team of humans to predict all potential situations in which the machine might find itself when operating autonomously in the real world. Therefore, a method is needed to augment the rule base automatically when new situations are encountered, or when exceptions to general rules are found. Induction of specific rules performs that role.
4. Induction of specific rules solves the frame problem, to the extent that the relevant environmental states are represented in the system.

To clearly define the concept, Shoham [36] divides the frame problem into the *qualification problem* and the *extended prediction problem*, and defines each separately. The qualification problem is concerned with missing conditions in a rule. For example, to light a match, several conditions must be satisfied: the match must be dry; there must be sufficient oxygen; the wind speed must be low; etc. When writing rules, some of these conditions might be omitted because they are "common sense", but a machine without a common sense database requires explicit "frame axioms" which specify these limiting conditions, and the number of frame axioms may be too large to be practical in a realistic environment. When specific rules are induced from experience, however, *all* states are assumed important until experience demonstrates that the values of some states are unimportant. Thus, there is no false prejudice which may cause the activation of a rule when it does not apply.

The extended prediction problem is related to the qualification problem, but it refers to the effects of a rule instead of its conditions. In particular, it is the problem of determining how long an effect predicted by a particular rule lasts, before the affected states are changed by another rule. It is a problem which infects the rule base of logic-based planners which have no built-in concept of time. *Ceteris paribus* reasoning, assuming that a state remains unchanged unless it is explicitly mentioned in the effect of a rule, is one approach to solving the extended prediction problem. Rule induction in PRIME takes the opposite approach, assuming that *all* states can be affected by an action, until experience shows that some states always remain unchanged after the execution of an action.

Therefore, specific rule induction solves both forms of the frame problem, but only if the necessary states are represented in the system. If this is not the case, then the use of probabilistic rules may overcome this deficiency.

1.3.3. Induction of general rules

Finally, PRIME must also induce general rules from the specific rules. *General rules* are rules which can be activated in more than one situation, or which can represent the effects of a class of actions, or both. Since specific rules only summarize what the machine has already experienced, general rules are necessary in order to apply the machine's knowledge in previously unencountered situations. They provide a way of extrapolating trends in the specific rules, in order to broaden the domain of the internal model.

Hence, the general approach of rule development in PRIME is *bottom-up*: specific rules are induced from experience, and general rules are induced from specific rules. By grounding its rules in experience, the machine circumvents some of the difficulties encountered in prefabricated rule bases, such as the frame problem, conflicting rules and overgeneralization—problems which are symptomatic of top-down development of rules,

but which can be avoided in bottom-up approaches. This methodology also allows the machine to constantly tune its rule base to maximize the accuracy of its internal model of the environment.

2. Literature Review

2.1. Rule representation and learning

This section briefly describes four existing paradigms for the representation and/or learning of action rules, and discusses the merits and shortcomings of using each in PRIME.

2.1.1. Predicate calculus planners

Predicate calculus planners are a large class of planners commonly used in Artificial Intelligence. The original concept was introduced by Green in [13], where he developed a planner which uses predicate calculus notation and resolution to construct plans which transform the initial environmental state to a state which satisfies a goal. The state of the world, the goal, the action descriptions and the invariant state relations are all expressed in predicate calculus well-formed-formulas (wff). Resolution is then used to prove the existence of a plan, and, as a side effect, the plan itself is derived. Green's method is sound (it finds only correct plans) and complete (it will find a correct plan if one exists), but it is highly inefficient and it suffers from the frame problem.

To address the frame problem, Fikes and Nilsson developed another planning system called STRIPS [10]. Representation in STRIPS is very similar to the wff representation in Green's method. The world model (or state of the environment) is expressed as a set of wff, and the goal is another wff. However, the operator descriptions are composed of three parts: a precondition (a formula which is true if the operator can be applied), an add list and a delete list. The two lists specify which wff are added and which are deleted from the world model after the action is executed. By using these lists to specify the effects of

an action, STRIPS avoids the extended prediction problem, but it is still faced with the qualification problem.

STRIPS may also be more efficient than Green's method because it uses *means-end analysis* [8] to generate a plan instead of straight resolution. In means-end analysis, a difference clause is generated which represents the disparity between the current world model and the goal. Then operators are chosen which, when applied, reduce the size of this difference clause. It then calls itself recursively twice: the first time to reduce the difference between the initial world model and the precondition of the operator; the second time to reduce the difference between the modified world model (modified by the operator) and the goal. It is therefore a divide-and-conquer strategy which generates subgoals based on the objective of bringing the environment state closer to the goal.

By using means-end analysis, however, STRIPS becomes incomplete, i.e., it is not guaranteed to generate a plan to satisfy the goal, even if one is known to exist. In particular, if the agent must take actions which move the environment state farther from the goal before the goal can be achieved, then the plan might not be found by means-end analysis. In addition, it has been demonstrated [37] that, under certain conditions, STRIPS may even be unsound, in the sense that it may generate incorrect plans.

Extensions and modifications of STRIPS have been developed to increase the efficiency or functionality of the basic system, e.g. MACROPS [9], ABSTRIPS [29], NOAH [30], and MOLGEN [39]. Nevertheless, all of these systems have three problems which prevent their use in PRIME. The first is that complete knowledge of the domain is required to generate plans, and this knowledge is encoded by a knowledge engineer (or team of knowledge engineers). For realistic environments, the knowledge base suffers from a combinatorial explosion (to a greater extent in the more complex modifications of STRIPS, since there is more information to encode), and the chance of error or omission

increases accordingly. The rules in PRIME can also suffer from a combinatorial explosion, but this does not increase the design effort, as it does in top-down systems such as STRIPS. The second problem, related to the first, is that there is no provision for rule induction or modification, which is necessary to correct errors and omissions on-line. Work has been done in generating new rules from existing ones [1, 4, 6], but no method exists for adding new rules or modifying old ones based on experience. The third problem is that there is no mechanism for representing the possibility of differing (perhaps even contradictory) effects for the same action under the same conditions. This is again related to the first problem, since this would be unnecessary if the machine had knowledge of all states.

2.1.2. Stochastic learning automata

A *stochastic automaton* is a unit which has the following elements:

- a set of internal states;
- a set of actions (or outputs), whose size is not greater than the state set;
- a probability vector which determines the random selection of the internal state at each time step;
- an update scheme which generates the next probability vector from the current probability vector, input and state;
- an output function which selects an action based on the current state.

A *stochastic learning automaton* (SLA) is a stochastic automaton whose update scheme is designed to minimize some cost function (or equivalently, to maximize some payoff function). For most SLA, the input is binary, representing either reward or penalty, and the output function is deterministic and one-to-one, so the action set is equivalent to the state set. The automaton operates in a closed loop with a stochastic environment, which

chooses a reward or penalty response randomly, based on the action dictated by the automaton and an unknown probability distribution. The objective of the SLA is to choose the action which minimizes the expected value of a penalty response from the environment. A good review of update schemes, their convergence properties, and applications of SLA can be found in [24].

Stochastic learning automata represent a completely different approach to the problem of selecting the proper actions to perform. It might initially appear that, since this is a probabilistic model operating in a stochastic environment, SLA could be useful in PRIME; however, the drawbacks outlined below eliminate SLA from consideration.

The first difficulty is that traditional SLA takes no input aside from the reward/penalty (or reinforcement) signal; it has no knowledge of the environmental state. To use SLA for selecting the proper action to perform, a separate automaton is needed for each ordered pair of environmental state vector value and goal vector value, which is impractical for two reasons. First, the number of automata increase exponentially with the size of the state vector and goal vector. As an example, for a reasonably sized binary state vector of 100 elements, the number of automata exceeds 10^{30}. Second, this configuration does not allow generalization over states or goals, which makes it impossible to apply the machine's knowledge in situations even slightly different from those previously encountered.

Barto and Anandan [3] have developed an *associative* SLA model, which takes a contextual input as well as a reinforcement, and uses the associative reward-penalty ($A_{R\text{-}P}$) algorithm as an update scheme. $A_{R\text{-}P}$ permits the learning of input-output mappings in environments which respond with only a reward/penalty signal to indicate whether the output chosen by the automaton was correct—this is called *associative reinforcement learning*. However, this algorithm has only been studied for the case of choosing between two actions.

Another difficulty with using SLA is that, in the environment in which PRIME operates, the reward signal would represent the achievement of the goal, and it needs to reinforce the entire sequence of actions, not just the last one. The problem of rewarding those actions in the sequence which actually contributed to the achievement of the goal, and penalizing all others, is called the *temporal credit assignment* problem, and SLA (associative and nonassociative) are not equipped to handle it. This problem is a consequence of the fact that the SLA is a reactive system: it does not predict the effects of its actions and it does no planning; it simply operates in a feedback loop with its environment, issuing actions in response to the environment's response, and hoping to receive some reward. The temporal credit assignment problem also appears in neural networks and classifier systems, and it is the reason why planning is necessary in PRIME.

2.1.3. Neural networks

"Neural networks" (also called "connectionist networks") is a collective term for a class of computation devices which have the following properties:

- they have a set of *nodes* (or neurons), each of which holds one value, sometimes called the output, which may be binary or real-valued;
- the nodes are connected with *links*, each of which holds one real value, called the *synaptic weight*;
- the information stored in the network is distributed among the weights;
- the weights are somehow used to determine the node output values.

Specifications beyond these are determined by the type of network model. In most models, the output of a node is computed locally by a weighted sum of other node outputs (the weights determined by the links connecting the nodes in question), passed through a simple nonlinear squashing function. Because these computations are local to each node, they are

highly parallel. Some models have specially designated input nodes and output nodes (the undesignated nodes called hidden nodes), while others allow any node to be used as input or output. Most models also have associated training algorithms for incrementally adjusting the weights of the network, such as back-propagation or Hebbian learning, but some models use a method such as outer-products to calculate a static set of weights. An overview of different models and training algorithms can be found in [2, 19, 28].

One of the advantages of neural networks is their massively parallel computation, which makes neurocomputing a viable alternative to traditional sequential approaches in problems which involve real-time processing of large amounts of data, such as signal or image processing. Another advantage is their distributed representation of knowledge which apparently results in a natural generalization ability; this leads to such applications as pattern classification, concept acquisition, associative memory, and applications which require noise immunity or other types of robustness.

There are several ways in which a neural network could be used in PRIME. For example, a network could take the environmental state and the goal as its input, and produce an action as its output—in particular, the action with the highest probability of leading toward the goal. This network would be trained by associative reinforcement, which poses the same temporal credit assignment problem discussed in the previous section. Williams [42] introduced a class of associative reinforcement learning algorithms called *REINFORCE* algorithms, which consist of two types: the restricted REINFORCE algorithm which receives immediate reinforcement, and the extended REINFORCE algorithm which receives delayed reinforcement and handles temporal credit assignment. Williams shows that several SLA update schemes are members of the REINFORCE class of algorithms. Unfortunately, networks using the REINFORCE weight update schema have three degrees of freedom (the learning rate, the reinforcement baseline and the out

probability mass function), and there is very little analytical or empirical guidance for their selection—no guidance at all in the case of the extended REINFORCE algorithms.

Another use of a neural network is to predict the effects of a particular action under a particular environmental state. This network could be trained with a supervised learning algorithm, in which the full input and output patterns are given in training. In general, it is easier to train a network with supervised learning than with reinforcement learning, because more information is provided to the network and patterns are more readily discriminated.

As a third type of application, some research has been done to develop neural networks which emulate symbolic processing capabilities, such as storage and recall of production rules [41], slot filling and role-binding in knowledge schemata [7, 40], and rule-based reasoning with variable binding [35]. However, none of these systems has the ability to learn new rules.

In the first two connectionist applications, the networks exhibit specific rule induction *and* general rule induction—due to the distributed representation of the information gained through experience, extrapolation (or interpolation) of this information occurs when the network is presented with a new input. Thus, it appears that neural networks may have some application in PRIME. In fact, Moed [22] has proposed a network architecture which operates in the Organization Level, learning the effects of actions and the probabilities of these effects. Nevertheless, it is meant to work in conjunction with a symbolic rule base containing specific and general rules. One disadvantage to using neural networks (as described in the first two applications) to exclusively represent the rule base is that they are difficult to initialize with *a priori* knowledge, if such knowledge is available. Neural networks must be trained with the knowledge, and this might take significant time and effort, whereas knowledge could be easily compiled into a set of symbolic rules; of course,

in both cases, the knowledge must be refined enough to allow significant automation of the process.

2.1.4. Classifier systems

Classifier systems, described by Holland in [16, 17], are blackboard production systems in which many rules can be active at the same time. The rules have a simple structure which facilitates matching and also allows induction of new rules. The basic elements of a classifier system are a list of classifiers, a message list, an input interface and an output interface.

The message list is the blackboard of the system. It contains a variable number of messages, each of which is a binary string of fixed length k:

$$M = \langle m_1, m_2, \ldots, m_k \rangle, \quad m_j \in \{0,1\}.$$

The list of classifiers constitutes the rule base, each classifier equivalent to a production rule. The classifier consists of a finite number of conditions, followed by an action:

$$C = c_1, c_2, \ldots, c_r / a \quad (r \geq 1)$$

where each condition and action is a trinary string of length k:

$$S = \langle s_1, s_2, \ldots, s_k \rangle, \quad s_j \in \{0,1,\#\}.$$

A message M matches a string S if the following two conditions hold for all $j \leq k$:

1. if $s_j = 0$ or $s_j = 1$ then $m_j = s_j$
2. if $s_j = \#$ then m_j can be either 0 or 1.

Thus the symbol # in a condition signifies a "don't care." The condition of a classifier is *satisfied* if each condition c_i is matched by some message on the message list. If this is the case, the classifier generates a new message from the action part, $a = \langle a_1, a_2, \ldots, a_k \rangle$, and the message $M = \langle m_1, m_2, \ldots, m_k \rangle$ which matched the condition c_1. The j^{th} element of the new message is given by

1. a_j, if $a_j = 0$ or 1
2. m_j, if $a_j = \#$.

Thus the symbol # in an action signifies a "pass-through." For example, if a classifier is given by $C = 01\#1\#0/11\#000$, and it is satisfied by a message $M_1 = 011100$, then it generates the outgoing message $M_2 = 111000$.

The purpose of input and output interfaces of the classifier system is to allow the system to communicate with the outside world with the same data structure used to represent internal information. The input interface places messages from the environment on the message list, while the output interface traps particular types of messages on the message list and sends them to the environment.

The basic execution cycle of a classifier system is given below:

1. Place all messages from the input interface on the current message list.
2. Compare all messages to conditions of all classifiers and record all matches.
3. For each classifier with all conditions satisfied, generate a message for the new message list.
4. Replace the current message list with the new message list.
5. Process the new message list through the output interface to produce system output.
6. Go to step 1.

One of the advantages of classifier systems is that they easily admit a parallel implementation of the basic cycle above, since classifier satisfaction (or rule activation) is independent of the order of the classifiers.

It should be emphasized that not all messages posted by the classifiers need be interpretable as output by the system; most are probably internal messages which serve the same role as assertions in the fact base of an expert system—they trigger the activation of other classifiers. Parts of the message string may even be reserved as *tags* recognized by

the conditions of classifiers, so that a message can be addressed to a particular subset of classifiers in the next cycle. This coupling of classifiers is a convenient way to build chains of classifiers which generate a sequence of actions.

Classifier systems may be extended to modify themselves by reinforcement learning. This requires two separate learning algorithms: one to apportion credit to the rules which are useful to the system, and one to generate new rules. The first learning task is performed by the *bucket-brigade algorithm*, and the second is performed by the *genetic algorithm*.

To accommodate the bucket-brigade algorithm, the basic classifier system must be modified. Associated with each classifier is a *strength*, which serves as a measure of the usefulness of the classifier in the past. An element of competition between classifiers based on their strengths is added to the basic execution cycle of the system, as follows. When all of a classifier's conditions are satisfied, it makes a *bid* based on its strength, its specificity (determined by the number of non-# symbols in the condition of the classifier) and its support (determined by the strengths of the classifiers which posted the matching messages). Now, instead of all satisfied classifiers being activated and posting their messages, only the highest bidding classifiers are activated. If a classifier wins a bid, it posts its message and it reduces its strength by its bid. Meanwhile, all the classifiers which posted messages matched by this winner have their strengths increased by a fraction of the bid, such that the bid is equally distributed among the supporting classifiers. Strength is essentially treated as a kind of capital, which is paid by active classifiers to supporting classifiers in the past. Eventually, a classifier might receive a payoff directly from the environment if it issues a correct command—this is where the reinforcement signal enters the system. In the long run, the strength of a classifier is increased if it tends to lead to a reward from the environment in the future, and it is decreased if it does not.

Incidentally, the specificity of a classifier is included in its bid to allow the existence of *default hierarchies*. A default hierarchy consists of general rules with more specific rules (ones with fewer #'s) below, serving as exceptions to the general rules (or defaults). These more specific rules may, in turn, have even more specific exception rules. Certainly, the exceptions (if strong) should be activated in place of the defaults; hence, preference is given to the more specific rules in the bidding process.

The genetic algorithm, introduced in [15], is a search technique which uses "genetic operators," such as crossover, to randomly generate new strings (or genotypes) from old ones. When applied to classifier systems, the classifiers are treated as genotypes. Parent classifiers are selected based on their strength, offspring are produced by genetic recombination, and these offspring replace the weakest classifiers. One of the advantages of using the genetic algorithm to create new classifiers is that useful building blocks (or string segments) within the classifiers tend to be retained.

Wilson and Goldberg [43] discuss many of the problems inherent in classifier systems with the bucket-brigade and genetic algorithms, such as the difficulty in generating and maintaining long chains of classifiers, overgeneralization, and reluctance to form default hierarchies. They also review some modifications to the two algorithms which are aimed at reducing these problems; these modified algorithms have met with varying degrees of success.

Zhou [44] discusses an important issue in using classifier systems for goal-directed behavior. He claims that the rules and chains of rules useful for achieving one goal may not be the same ones useful for achieving another; therefore, it is necessary to keep separate classifier sets for each goal to prevent useful rules from losing strength when they are applied in the wrong context. However, this prevents generalization across goals and increases learning effort. In the context of PRIME, this problem is easily solved by placing

the goal (expressed as a binary string) on the message list, and letting the classifiers condition on parts of the goal string.

Classifier systems present a unified approach to the problems of rule learning and goal-directed behavior. They exhibit both specific and general rule induction. They maintain rule strengths which may be used to evaluate the performance of the system. The rule representation is simple and it permits fast condition matching.

Still, the problems of long chain generation and maintenance remain. In addition, problems may arise with using the knowledge base to achieve previously unseen goals, if the rules are conditioned on the goal. It would be more desirable to model the effects of the systems actions explicitly, so the system could plan with a common rule base to achieve any given goal.

The above considerations preclude the use of classifier systems in PRIME, but they do not eliminate the advantages of the rule representation within these systems. For this reason, the representation developed in §3.2.2 is modelled after the classifier.

2.2. Probability estimation

2.2.1. Requirements

Section 1.3.1 established the need for an algorithm which estimates, for a given condition and action, the probability that a particular effect occurs, i.e., the probability that some environmental state description obtains after the action is executed. This problem can be viewed in two ways: as the estimation of a discrete probability distribution over effects; or as the independent point estimation of the probabilities of several effects, with the constraint that the sum of all the estimates must be one.

Regardless of which approach is used, there are several requirements of the estimation procedure which must be fulfilled:

1. it must be recursive (or sequential);
2. it must assume no *a priori* knowledge about the probabilities, although initial probability values should be used if provided;
3. there must be no smoothing of the distribution, since the ordering of effects is arbitrary;
4. it must be able to track nonstationary probabilities.

Sequential estimation is necessary because the estimates must be updated on-line, during the machine's operation, to ensure the accuracy of the estimates. To keep PRIME as general and flexible as possible, no assumptions are made about the environment except that it is a Markov process, and even this may be relaxed in the future (see §6.5). This requirement eliminates not only parametric estimators, which assume a structure for the distribution function being estimated, but also Bayesian estimators, which assume a structure for the PDFs of the probability estimates [12]. The fourth requirement, the ability to track time-varying probabilities, implies that some variance is necessary in the probability estimates, forcing the estimator to balance the needs of accuracy and stability.

The next three sections explore the relationships between three estimation procedures which are used (with some modification) in PRIME, as described in §3.3.2.

2.2.2. Sample mean

For a given condition and action, let N be the number of effects or next states which can result from the execution of that action under that condition. Let the vector of physical (or real) probabilities of those effects be $Q = (q_1, q_2, \ldots, q_N)$. Whenever the environment receives the action in question under the specified condition, it chooses a next state from a

vector of effects $(e_1, e_2, \ldots, e_N)$ randomly, using Q. This feedback from the environment can be expressed in the form of a random vector $(x_1, x_2, \ldots, x_N)$, where for $1 \le i \le N$:

$x_i = 1$ if e_i was chosen,

$x_i = 0$ otherwise.

Thus, $q_i = P\{x_i = 1\}$.

Let the estimate of each q_i at step n be $p_i[n]$. Since x_i is a Bernoulli random variable, $E\{x_i\} = q_i$. Thus, q_i can be estimated by the sample mean:

$$p_i[n] = \frac{n_i}{n} = \frac{1}{n}\sum_{j=1}^{n} x_i[j]$$

This can be put in recursive form:

$$p_i[n+1] = \frac{n}{n+1}p_i[n] + \frac{1}{n+1}x_i[n+1] \qquad (2.1)$$

Note that this sequence converges to a constant value, since the second term $\rightarrow 0$ as $n \rightarrow \infty$. In fact, the strong law of large numbers states that $p_i[n]$ converges to q_i almost everywhere, i.e.,

$$P\{\lim_{n\rightarrow\infty} p_i[n] = q_i\} = 1$$

This is actually a detriment, because we need an estimator which can respond to changes in q_i. This becomes harder to do as time progresses, due to the decaying weight of new data in (2.1). The weight of new data could be fixed by using a "moving window" approach—taking the sample mean of the last w points, where w is a constant. It can be shown that the variance of $p_i[n]$ is inversely proportional to w, allowing the adjustment of w to trade off between tracking ability and noise level. Nevertheless, the moving window approach is not recursive, and it requires the storage of w points of data. In addition, the same tradeoff can be accomplished with the linear reinforcement scheme, §2.2.4.

2.2.3. Stochastic approximation

The Robbins-Monro stochastic approximation algorithm [27, 31] was originally presented as a method of searching for the unique root of an unknown function, with the function output corrupted by additive, zero-mean noise. Specifically, if

$$y(\phi) < \infty \quad \forall \phi \in [a,b], \text{ and}$$

$$y(\theta) = 0 \text{ for some unique } \theta \in [a,b],$$

and the noisy output is given by

$$z(\phi) = y(\phi) + v$$

$$E\{v\} = 0$$

$$E\{v^2\} < \infty,$$

then the sequence

$$\phi[n+1] = \phi[n] - \gamma[n] z(\phi[n])$$

converges to θ almost everywhere if the sequence $\{\gamma[n]\}$ satisfies the following conditions:

$$\lim_{n\to\infty} \gamma[n] = 0$$

$$\lim_{n\to\infty} \sum_{k=1}^{n} \gamma[k] = \infty$$

$$\lim_{n\to\infty} \sum_{k=1}^{n} \gamma^2[k] < \infty$$

To cast this procedure in terms of probability estimation, let

$$\theta = q_i$$

$$\phi = p_i$$

$$y(\phi) = \theta - \phi = q_i - p_i$$

$$v = x_i - q_i$$

$$z(\phi) = y(\phi) + v = x_i - p_i$$

so the estimation sequence becomes

$$p_i[n+1] = p_i[n] + \gamma[n](x_i[n] - p_i[n]). \tag{2.2}$$

Now, if we let $\gamma[n]$ be the harmonic sequence

$$\gamma[n] = \frac{1}{n+1}$$

then the stochastic approximation (2.2) becomes the recursive sample mean estimation (2.1). Thus, sample mean is a special case of Robbins-Monro stochastic approximation. Other choices exist for $\gamma[n]$, of course, but since stochastic approximation is guaranteed to converge almost everywhere, it suffers the same problem as sample mean—it ability to adapt to a nonstationary environment decays with time.

2.2.4. Linear reinforcement

Recalling the discussion of stochastic learning automata in §2.1.2, the problem of learning the probabilities over possible effects of a given rule's action may be compared to learning the action probabilities in an automaton, where the automaton's action corresponds to predicting the effect of the rule's action. In this section, the linear type of update scheme is investigated, as it pertains to learning effect probabilities in PRIME.

In general terms, the essence of any automaton reinforcement scheme is this: an action a_i is selected; if the environment responds with a reward, the probability p_i of selecting that action is increased, and all other probabilities are decreased; if the environment responds with a punishment, p_i is decreased and all other probabilities are increased. In the case of the *linear reward-penalty* ($L_{R\text{-}P}$) scheme [24, 42], the following equations govern the update:

if action a_i is rewarded:

$$p_i[n+1] = p_i[n] + \alpha(1 - p_i[n]) \tag{2.3a}$$

$$p_j[n+1] = p_j[n] - \alpha p_j[n] \quad \forall j \neq i \tag{2.3b}$$

if action a_i is penalized:

$$p_i[n+1] = p_i[n] - \beta p_i[n] \tag{2.3c}$$

$$p_j[n+1] = p_j[n] + \beta(\frac{1}{N-1} - p_j[n]) \quad \forall j \neq i \tag{2.3d}$$

where $\alpha, \beta \in (0,1)$ are learning rate constants, and N is the number of possible actions. If $\beta=0$, the update scheme is called *linear reward-inaction* (L_{R-I}), since no correcting action is taken in the case of a penalty response. L_{R-P} and L_{R-I} can be collectively called *linear reinforcement* schemes.

An automaton representing a collection of rules with identical conditions and actions could be trained by reinforcement. The training cycle would operate as follows:

1. The automaton predicts an effect based on the vector of internal effect (or action) probabilities.
2. After the action part of the rule is executed, the effect (or next state) is observed. If the effect was correctly predicted, a reward signal is sent to the update scheme, else a penalty signal is sent.
3. The update scheme changes the effect probabilities based on the reward/penalty signal and the chosen effect.

Although these linear reinforcement schemes demonstrate different convergence behaviors (L_{R-P} is expedient and L_{R-I} is ε-optimal), the convergence destination which these and many other SLA update schemes strive for is the same: to maximize the expected value of the reward signal by choosing the proper action. Since, according to the model, the environment randomly chooses to reward an action based on a fixed set of reward probabilities for each action, an "optimal" automaton would always choose the action with the highest reward probability. It follows that the update schemes strive to converge the action probability vector to a vector of 0's except for the element corresponding to the optimal action, which is 1. Therefore, the linear reinforcement schemes would *not*

converge to the correct effect probabilities if the automaton representing the rule were trained by reinforcement, as outlined above.

However, within PRIME, the effect probabilities need not be learned by reinforcement; they can also be developed by supervised learning. The training cycle modified for supervised learning is:

1. The action part of the rule is executed, and the effect is observed.
2. The probability corresponding to the resulting effect is increased, while all other probabilities are decreased.

In this supervised learning mode, as opposed to the reinforcement learning considered before, the automaton does not actively predict the effect of the rule's action; rather, it is given the effect, and the update scheme simply updates the effect probabilities. This may be considered equivalent to a situation where the automaton always correctly predicts the effect, because it may act after the answer is known. Therefore, there is no penalty signal, and only equations (2.3a-b) are necessary—$L_{R\text{-}P}$ and $L_{R\text{-}I}$ become indistinguishable. The term *linear reinforcement* will henceforth refer only to equations (2.3a-b).

Appendix A (equations (A.5) and (A.9)) shows that, for the linear reinforcement scheme,

$$\lim_{n\to\infty} E\{p_i[n]\} = q_i$$

$$\lim_{n\to\infty} \sigma_p^2[n] = \frac{\alpha}{2\text{-}\alpha}\sigma_x^2$$

where $\sigma_x^2 = q_i(1\text{-}q_i)$ and α is the learning rate constant. This implies that the estimated probabilities converge in mean square to the physical probabilities if $q_i \in \{0,1\}\ \forall i$; if this is the case, however, the environment is deterministic and there is no estimation, only storage of the single possible effect. Nonetheless, these limits also imply that the estimates might converge in mean square for $q_i \in (0,1)$ if $\alpha \to 0$ as $n \to \infty$, although the equations were

derived assuming α is constant. They also demonstrate that, because the variance of the estimate is nonzero for a stochastic environment even in the limit, linear reinforcement will always adapt to changes in the physical probabilities, although this variance should be controlled by decreasing α.

Linear reinforcement is intimately related to recursive sample mean and stochastic approximation. Consider a modified version of recursive sample mean, in which the effect of a particular datum on the probability update decays over time; this counteracts the hard convergence property by weighting newer data more heavily than older data. If an exponential decay is used, the modified recursive sample mean is:

$$p_i[n+1] = \frac{\beta n}{n+1} p_i[n] + \frac{(1-\beta)n + 1}{n+1}$$

$$= 1 - \frac{\beta n}{n+1}(1-p_i[n])$$

$$p_j[n+1] = \frac{\beta n}{n+1} p_j[n] \quad \forall j \neq i$$

for $x_i = 1$ and $\beta \in (0,1)$. For large n,

$$p_i[n+1] \approx 1 - \beta(1-p_i[n]) = (1-\beta) + \beta p_i[n]$$

$$p_j[n+1] \approx \beta p_j[n] \quad \forall j \neq i$$

If $\alpha = 1 - \beta$, then

$$p_i[n+1] \approx \alpha + (1-\alpha)p_i[n] = p_i[n] + \alpha(1-p_i[n])$$

$$p_j[n+1] \approx (1-\alpha)p_j[n] = p_j[n] - \alpha p_j[n] \quad \forall j \neq i$$

But these are the equations for linear reinforcement (2.3a-b)!

Now suppose linear reinforcement had a decaying learning rate, to allow it to converge to constant estimates. In condensed notation, the update equation becomes

$$p_i[n+1] = p_i[n] + \alpha[n](x_i - p_i[n]) \quad 1 \leq i \leq N$$

which is identical to (2.2), the update equation for stochastic approximation.

Section 3.3.2 develops a method of using recursive sample mean and linear reinforcement concurrently, to exploit the advantages of both estimation schemes.

3. Design of PRIME

3.1. Architecture of PRIME

The purpose of this section is to describe the basic features of the architecture of PRIME; the details of these features will be presented in the remaining sections of this chapter. This architecture was designed to satisfy the functional requirements of the system given in §1, which are summarized as follows:

1. to dictate high-level actions to a system of machines (such as robots) in an abstract environment with minimal *a priori* knowledge;
2. to maintain a probabilistic internal model of the environment, in the form of a rule base;
3. to develop this rule base in a bottom-up fashion, inducing specific rules from experience, and inducing general rules from specific rules;
4. to plan with the rule base to achieve a user-specified goal with maximum probability of success.

A block diagram illustrating the major components of PRIME appears in Figure 3.1. However, the following sections of this chapter, for the most part, do not discuss the modules designated in the diagram; rather, they specify the operation of the three major processes in PRIME: *exploration*, *generalization* and *planning*.

Exploration is a process which allows PRIME to explore its environment by executing actions, observing their effects, and forming new rules or updating old ones with this experience; it is the primary method for specific rule induction. In exploration, the rule base gives a set of active rules to the explorer, based on the current state of the environment The explorer picks one of these rules to test or creates a new one, and feeds the rule to the

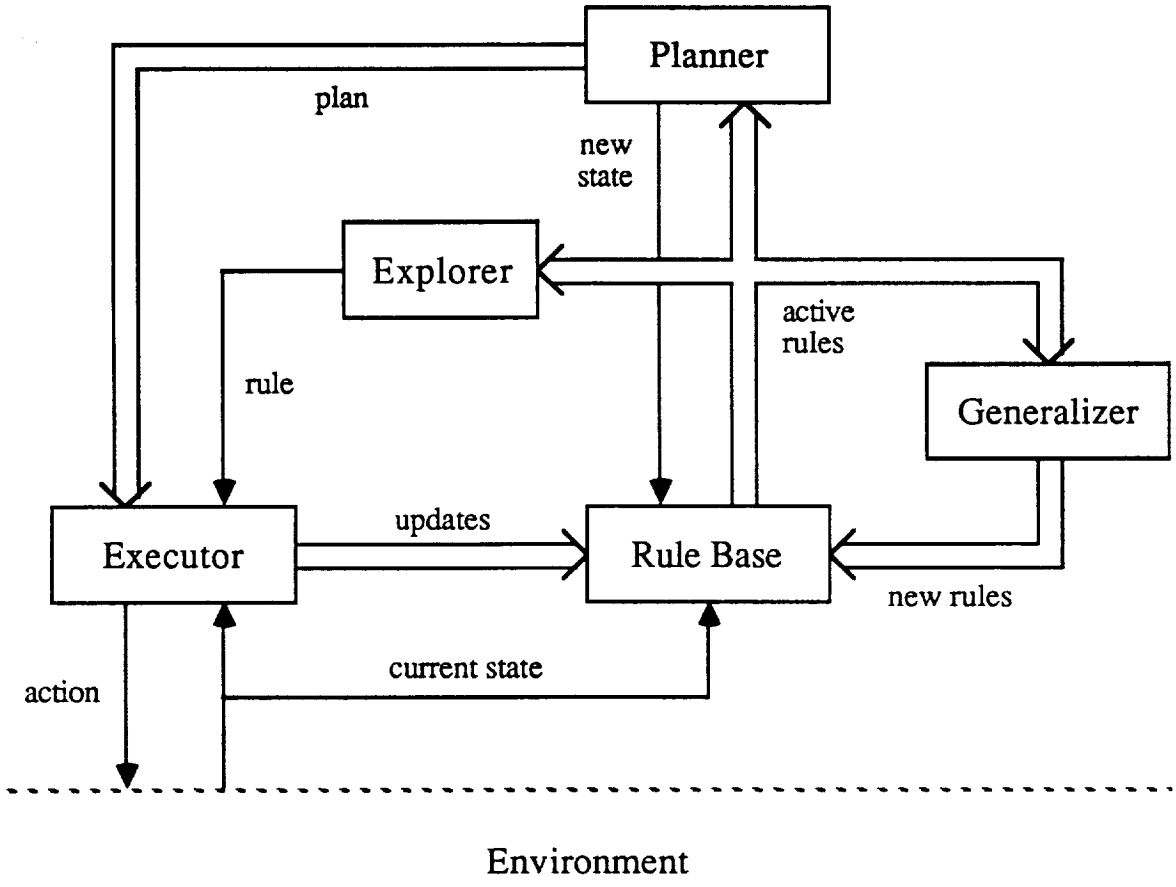

Figure 3.1 PRIME block diagram

executor. The executor sends the action to the environment, which responds with a new environmental state, and the executor updates the rule base with this new information.

In generalization, the active rules for a given state are used to create new general rules, which are then added to the rule base. This process can run in parallel with either exploration or planning.

Planning involves a cycle of interaction between the planner and the rule base. During its operation, the planner needs to explore the effects of actions under different environmental states which may occur during the execution of a sequence of actions; this is

accomplished by repeatedly sending a new state to the rule base and examining the resulting active rules. The ultimate output of the planner is a plan, which internally is represented by a list of rules which contain the actions to execute and their expected effects. The executor sends each of these actions to the environment in turn, and uses the environmental state observed after the execution of each action to update the rule base, just as in exploration.

Since the division into modules illustrated in Figure 3.1 is rather artificial, the processes themselves will be described in the subsequent sections. First, however, the representation of the environment and the rules must be presented.

3.2. Representation

This section defines the internal representation of the abstract environment and the rules in PRIME, which were briefly characterized in §1.2 and §1.3.1. To illustrate the concepts, examples are derived from the "box and room" world model which was used to test the STRIPS system [10]. However, the application of PRIME is by no means limited to this type of "toy problem"—a more realistic environment, based on NASA's Flight Telerobotic Servicer [25], is developed in §4.5 for demonstrating the capabilities of PRIME.

3.2.1. Environment

PRIME's representation of the environment consists of a set of abstract *states*, sets of *actors*, *verbs* and *objects*, and sets of *actor classes* and *object classes*, all of which are defined in advance. The actor set, such as

$$\mathbf{A} = \{\text{robot1}, \text{robot2}, \text{robot3}\}$$

is the set of agents which PRIME uses to cause changes in the environment. The verb set, such as

V = {gotoloc, gotoobj, pushto, turnonlight, climbonbox, climboffbox, gothrudoor}

is the collection of commands which PRIME can give to each of the actors. The set of objects, such as

U = {box1, box2, box3, room1, room2, room3, room4, room5, door1, door2, door3, door4, lightswitch1, loc_a, loc_b, loc_c, loc_d}

includes both the material objects which the actors can affect, and the immaterial objects which do not change but which qualify the action to be executed. Some verbs take objects in the action sentence; for instance, the verb pushto takes two objects, both of which must be boxes, while gotoloc takes one object, which must be a location. A description of the syntax of an action sentence is given in the next section.

An actor class is a subset of the actor set **A**; a sample set of actor classes is:

$\mathbf{C}_A$ = {*lightrobot, heavyrobot*}

lightrobot = {robot1, robot2}

heavyrobot = {robot3}

Likewise, an object class is a subset of **U**; a sample set of object classes is:

$\mathbf{C}_U$ = {*box, room, door, lightswitch, loc, null*}

box = {box1, box2, box3}

room = {room1, room2, room3, room4, room5}

door = {door1, door2, door3, door4}

lightswitch = {lightswitch1}

loc = {loc_a, loc_b, loc_c, loc_d}

$$null = \varnothing$$

The *null* class is used as a placeholder in action sentences which do not use all of the object fields. To simplify implementation, the current version of PRIME forces these classes to be disjoint partitions of the actor and object sets, i.e.,

$$\cup \mathbf{C_A} = \mathbf{A}, \ \cap \mathbf{C_A} = \varnothing$$

$$\cup \mathbf{C_U} = \mathbf{U}, \ \cap \mathbf{C_U} = \varnothing$$

The state set is a set of propositions about actors and objects in the environment. A partial example of a state set is:

$$\mathbf{Z} = \{\text{box1\_at\_loc\_a, box1\_at\_loc\_b, box1\_in\_room1, ...,}$$
$$\text{box2\_in\_room1, box2\_in\_room2, box2\_in\_room3, ...,}$$
$$\text{robot1\_on\_floor, robot1\_on\_box1, robot1\_on\_box2, ...,}$$
$$\text{robot2\_at\_loc\_a, robot2\_at\_locb, robot2\_nextto\_box1, ...,}$$
$$\text{lightswitch1\_on}\}$$

The state set $\mathbf{Z}$ has N states, which can be ordered to form the vector $\mathbf{Z}' = (z_0, z_1, \ldots, z_N)$, and the truth value of this vector can be summarized by a binary state vector, or *state string*:

$$\mathbf{E} = (e_0, e_1, \ldots, e_N) \in \mathbf{B} = \{0,1\}^N$$

$$\text{where } \forall i,\ e_i = \begin{cases} 1 \text{ if } z_i \text{ is true} \\ 0 \text{ if } z_i \text{ is false} \end{cases}$$

The value of $\mathbf{E}$ at any given time will often be called the "state of the environment" or the "environmental state," which is not to be confused with the state set $\mathbf{Z}$ or an object state, which is a member of $\mathbf{Z}$.

3.2.2. Rule structure

Recall from §1.3.1 that a rule consists of a condition, an action, an effect and a probability. The precise structure is developed below.

An *action sentence* (or simply an *action*) is an M-tuple

$$\mathbf{S} \in \left((\mathbf{A} \cup \mathbf{C_A}) \times \mathbf{V} \times (\mathbf{U} \cup \mathbf{C_U})^{M-2}\right)$$

which consists of one actor or actor class, one verb, and M-2 objects. A common configuration might be $M = 4$, in which case the first object could be called the *direct object*, while the second could be called the *indirect object*. All actions are the same length, so M should be chosen to reflect the maximum number of required objects over all verbs. If a verb takes less than M-2 objects, the unused slots are filled with the *null* class internally, but these slot fillers need not be mentioned in textual descriptions. Examples of action sentences are:

robot1 pushto box1 box3

lightrobot climbonbox box2

robot3 gotoloc *loc*

heavyrobot gothrudoor *door* room5 *room*

Specific actions are actions which do not contain any classes; the first action above is a specific action. *General actions* are actions which contain at least one class; the last three actions above are general actions. Only specific actions are issued to the environment, but both specific and general actions appear in the rule base.

A *trinary string* is a string over a three-character alphabet with the same length as the state vector; it is a member of the set

$$\mathbf{T} = \{0,1,\#\}^N$$

The *condition string* and *effect string* (or just *condition* and *effect*) are both trinary strings. A *specific condition* contains no # symbols, whereas a *general condition* contains at least one # symbol. Now a *rule* can be defined as a quadruple, a member of the set

$$\mathbf{R} = (\mathbf{T} \times \mathbf{S} \times \mathbf{T} \times \Re)$$

so every rule consists of a condition, an action, an effect and a real-valued probability, in that order. A *specific rule* has a specific condition and a specific action; if the rule contains a general condition or a general action (or both), it is a *general rule.* A rule with a general condition applies in more than one situation, and a rule with a general action predicts the effect of more than one action.

A rule is *active* if its condition is matched by the state string **E**. This matching is identical to the matching of a classifier (§2.1.4): 0's in the condition match only 0's in the state, 1's in the condition match only 1's in the state, and #'s in the condition match 0's and 1's. Likewise, the effect (or next environmental state) predicted by an active rule is generated from the effect string in the same way that a new message is generated from the action string of a satisfied classifier: the new state string is the effect string with all #'s replaced by the corresponding value in the old state string. Thus, the # in the condition string acts as a wildcard, while the # in the effect string acts as a carry-through.

This matching and predicting process amounts to a method of *instantiation* of a rule with a general condition, in the context of the given state string. As yet, however, no mention has been made of the method for instantiating a rule with a general action, in order to predict the effects of a particular action with this rule. This will be discussed in the next section.

In general, the rule base is used as follows: Given a particular state string (which may or may not be the current environmental state), the set of active rules for this state is found. Each active rule predicts the effect on the environment of the execution of its action under the activating state, with a certain probability. However, for each specific action, there exists a hierarchy from general rules to specific rules which predict the effect of executing that action; this is the default hierarchy mentioned in §2.1.4. Since less general

rules may serve as exceptions to more general rules in this hierarchy, the least general (or most specific) rule for a given action is used to predict the new environmental state.

The decision to represent the conditions and effects of rules is supported by two factors: efficiency of storage and efficiency of recall. The state string (and therefore the condition and effect strings) in a realistic environment can become quite long, but the storage requirements of rules become manageable if the condition on each object state occupies only two bits, which is the case with trinary digits. Also, the matching procedure for finding active rules is simple and quick—PRIME uses a tree-like structure to store rules so that the time for condition matching is linear in the length of the condition string. If the parallel nature of this procedure were exploited in hardware, even faster recall could be possible.

3.2.3. Locality of effect

Locality of effect is the assumption that only the actor and objects mentioned in a given action sentence can be affected by that action; all other objects remain unchanged. This assumption is made in the current version of PRIME for two reasons: it condenses the condition and effect strings, and it allows the existence of general actions as defined in the previous section. Each of these reasons will be considered in turn.

To take advantage of locality of effect, a mask defined for each actor and object is added to the *a priori* environment specification. A *mask* is a binary string from the set **B** which is defined as follows:

$$\mathbf{M} = (m_0, m_1, \ldots, m_N) \in \mathbf{B}$$

$$\text{where } \forall i,\ m_i = \begin{cases} 1 \text{ if } z_i \text{ is relevant to the actor or object} \\ 0 \text{ if } z_i \text{ is not relevant to the actor or object} \end{cases}$$

Thus, the mask for an actor or object indicates which states can be affected by an action containing that actor or object. Since it is known that the states which are irrelevant to the actor and objects of a rule will remain unchanged, these states can be removed from the condition and effect strings of the rule.

When a new specific rule is generated, the condition is derived from the environmental state before the execution of the action, and the effect is derived from the resulting environmental state after executing the action. However, under the locality of effect assumption, it is unnecessary to store the whole state string in the condition and effect fields of the rule, because some states are irrelevant. It is only necessary to store those states which are deemed relevant by at least one of the pertinent mask strings. The condition and effect can be "compiled" into strings which are potentially much shorter than the state string. Now the trinary condition and effect strings are no longer members of $\mathbf{T}$; they have variable length, dictated by the mask strings of the actor and objects of the rule. Thus, rules are now members of the set schema

$$\mathbf{R}_K = (\mathbf{T}_K \times \mathbf{S} \times \mathbf{T}_K \times \Re)$$

where $\mathbf{T}_K = \{0,1,\#\}^K$ and $K \leq N$

Depending on the environment, this may achieve a great reduction in the storage requirements of the rule base.

Since the definition of a condition of a rule has changed, the definition of an active rule must change accordingly. A rule from the schema $\mathbf{R}_K$ is said to be *active* if all of the states which are relevant to the actor and objects of the action satisfy the requirements specified by the condition string. The individual matching of state values to condition values is the same as in the classifier system, but the entire state string can no longer be matched with the condition string in one operation, because their lengths and arrangement of states are different. However, the matching procedure is only slightly more

complicated, because the object masks (or an altered version of them) can easily be used to determine which state corresponds to each element of the condition string, and only those relevant states need to be tested.

Figure 3.2 provides an example of condition compilation; the same procedure holds for effect compilation. The masks of the actor and objects of the action robot1 pushto box1 box3 appear next to copies of the state string. The state values in positions which correspond to 1's in each mask are copied and concatenated to form the condition string. Notice that the condition string (and consequently the effect string) is effectively partitioned into *fields* which correspond to the states relevant to the actor and individual objects of the rule.

In order to develop rules with general actions (that is, with a class name used as an actor or object), it is necessary to keep the interpretation of the states in a given field invariant over all members of a class. This is achieved by constraining the design of the state set such that all members of a particular actor or object class have the same number and types of states. For instance, if the object box1 has the following relevant states:

box1_at_loc_a, box1_at_loc_b, ...,

box1_in_room1, box1_in_room2 ...

then box2 must have the corresponding relevant states:

box2_at_loc_a, box2_at_loc_b, ...,

box2_in_room1, box2_in_room2 ...

and all members of the class *box* must have similar states. If this constraint is met, then any condition field corresponding to a member of a particular class can be interpreted as a field corresponding to another member of the same class. This allows the field to correspond to the class name itself, which can later be replaced with any of its members to form a coherent rule with a specific action.

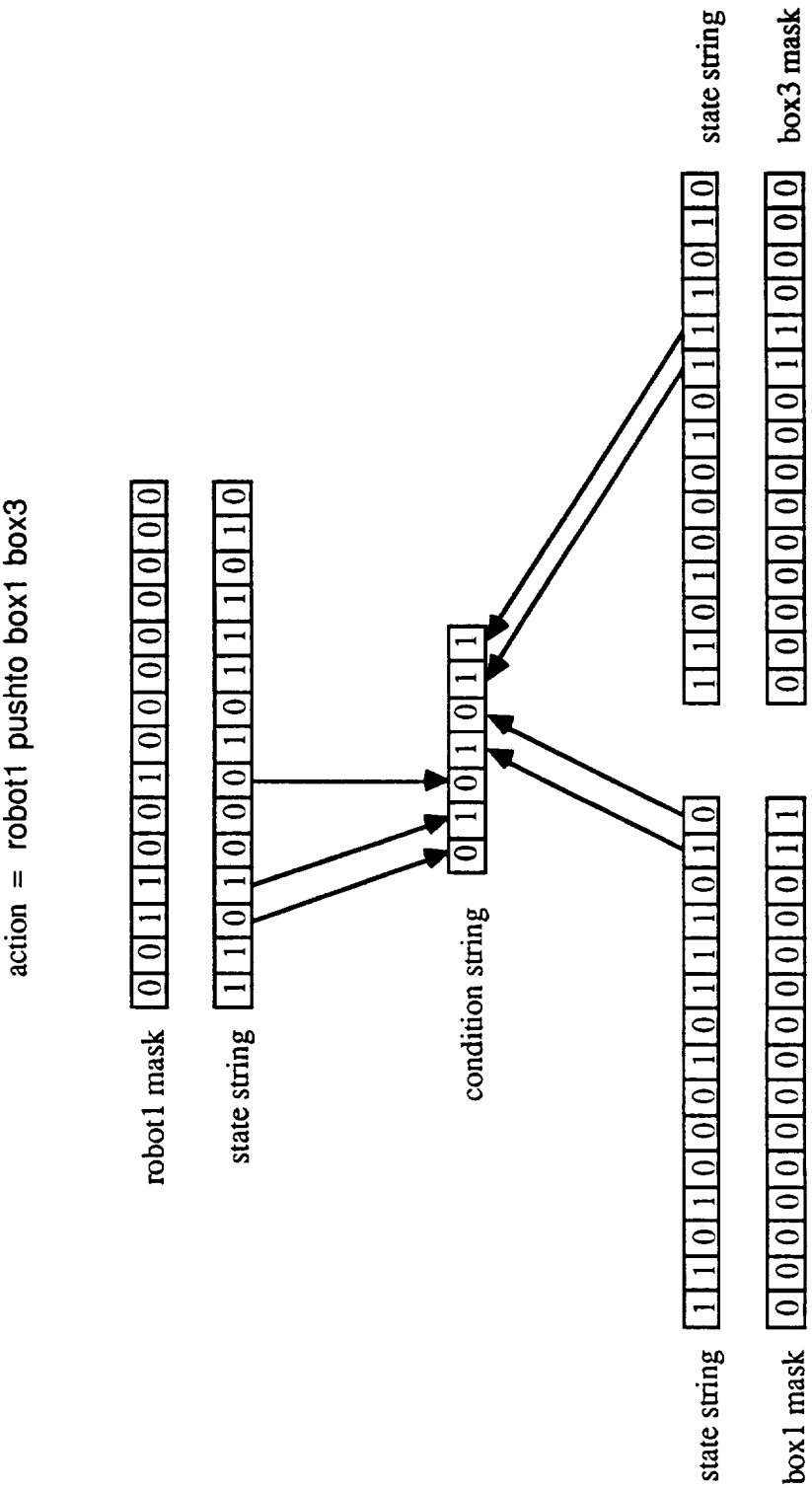

Figure 3.2 Compilation of condition string

To continue the example, suppose there is a rule for the action robot1 climbonbox box1 whose condition calls for box1 to be at loc_a in room2. Then the object field of the condition string might look something like (10...01...), if the state positions in the condition string correspond to the positions of the relevant states listed above. If there is another rule, whose action is robot1 climbonbox box2, and whose condition calls for box2 to be at loc_a in room2, then the object field of the condition string would look like (10...01...), identical to the previous field. These two rules could be generalized (the exact method is presented in §3.4.2) to form a general rule with the same condition but with the action sentence robot1 climbonbox *box*. This general rule can be instantiated by replacing the class name *box* with any of its members, and the condition field for the object can be interpreted according to the specific object chosen. So, if a situation occurs where box3 is at loc_a in room2, then this general rule can be activated with the action sentence instantiated as robot1 climbonbox box3.

In summary, masks provide a mapping between condition string fields and state strings, and these masks are related in such a way that the states pertaining to an actor or object of a particular class can be translated to states of similar meaning pertaining to any other actor or object of the same class. They furnish a simple way of implementing state predicates (states which take objects as arguments), without resorting to predicate calculus notation. This, in turn, allows the rule representation power to extend to general-action rules, which can predict the effects of a whole class of specific actions, even ones which have not been tested yet.

The cost of assuming locality of effect is the reintroduction of the frame problem. Recall from §1.3.2 that induction of specific rules avoids the qualification problem, to the extent that the necessary states for distinguishing different conditions are represented in the system. If states are uniformly eliminated by locality of effect, then it is possible that such

a distinguishing state may be overlooked. Consider a problem where a robot needs to pick up a box, say box1, but in order to pick up this box, a particular door must be open so the robot can get to the box. The state of the door will not appear in the condition of a rule with the action robot1 grasp box1, since the door does not appear in the action sentence. Thus, the action will appear to fail a high percentage of the time, and under the same conditions as when it succeeds. The solution is to add another state, such as box1_access_clear, which can then appear in the condition of the rule for robot1 grasp box1. This amounts to adding a frame axiom, which is an *ad hoc* solution, and it puts additional burden on the designer when developing the *a priori* environment representation. For all of its advantages, locality of effect is still a drawback in the long run; a method for eliminating the need for locality of effect should be a focus of future research.

3.3. Rule discovery

3.3.1. Exploration

Exploration is the process by which PRIME performs specific rule induction, or rule discovery. The purpose of exploration is to allow the machine to explore its environment and develop specific knowledge about the effects of its actions in specific situations. This is achieved through direct interaction with the environment: executing an action and observing the changes in the environmental state. The interaction is random, in the sense that the selection of actions to execute or rules to test is random.

The basic cycle of exploration is:

1. Search for the set of active rules for the current environmental state.

2. Generate a random action, or get the action from a randomly selected rule from the active set.
3. Execute the action, and observe the new environmental state.
4. If a specific rule which predicts the new state does not exist, create one.
5. Update effect probabilities among all active specific rules whose action sentence is the executed action.
6. Update probability and support statistics among all active general rules whose action sentence is the executed action, and delete any general rules which do not have enough support.
7. Go to 1.

A new rule is created as described in the previous section: the prior state string is compiled into the condition string, and the posterior state string is compiled into the effect string. In addition, if the chosen action has never been executed under the prior state, the effect probability is initialized to 1; otherwise, it is initialized to 0 and is updated along with the effect probabilities of the other rules (step 5). The probability estimation procedure described in the next section is used to update the effect probabilities of specific rules.

Each general rule keeps track of its *support*, which is the set of existing specific rules which are instantiations of the general rule, and whose effect probabilities are above a certain support threshold (e.g. 0.8). An *instantiation* of a general rule is a specific rule formed by replacing each # in the condition string by a 1 or 0, each # in the effect string by the corresponding value chosen in the condition string, and each class name in the action sentence by a member of the class. The effect probability in a general rule is simply the average of the effect probabilities of its supporting rules, so updating a general rule's probability in step 6 amounts to updating this average to reflect whatever updates were made to supporting rules in step 5. If a general rule gains or loses support due to the

updates in the probabilities of specific rules, the support count is also updated in step 6. If a general rule is then found to have too little support to justify its existence, it is deleted. Further discussion of the maintenance of general rules can be found in §3.4.

The role of exploration is not limited to refining the rule base. If no initial rule base is supplied, exploration within a simulated environment is used to generate the rule base from scratch. This can be an effective way of generating a rich set of rules automatically, but only if the machine is exposed to a variety of situations, and if "key" situations appear frequently enough. One problem with performing exploration in a closed loop like the one above is that the machine might fall into uninteresting situations too often, while failing to build up the rules which may be key to completing useful plans later. To rectify this imbalance, the exploration algorithm for initial training could be modified so that a new, random environmental state is presented before step 1 at each cycle, essentially breaking the loop. The probability distributions for generating this random state could then be tailored to provide sufficient exposure to important situations. However, this may entail a significant design effort, making it more expensive than manually generating an initial rule base.

In the normal operation of an intelligent machine in the field, the primary function of PRIME is planning. Nevertheless, as plans are executed, the effects of actions are observed, and specific and general rule induction still take place. So when is exploration necessary to refine the rule base? In general, a supervisor (human or machine) is needed to determine that the rule base is too incomplete to effectively plan under certain conditions. Another approach, though, might be to have PRIME detect situations where it has developed a partial plan, but cannot plan any further because of lack of information in the rule base; the machine could execute the partial plan and then explore for a while, until a sufficient number of rules have developed, or until it reaches a familiar state from which it

can plan further. These are topics for future research, however, and are not addressed in this work.

3.3.2. Probability estimation

The refinement of the effect probabilities of the rules in the rule base is an important function of exploration. Constantly updating these probability estimates during exploration or execution of plans allows the rule base to include the latest information available about the environment, which in turn allows it to adapt to changes. Maintaining the accuracy of the estimates ensures that the plans generated in planning represent the most optimal path of action. The following sections describe how PRIME updates the effect probability estimates by using two estimators, a short-term one and a long-term one.

3.3.2.1. Long-term and short-term estimates

In §2.2, the constraints on the probability estimation in PRIME were outlined, and three different but related methods were presented for estimating effect probabilities. The sample mean estimator, which is a special case of Robbins-Monro stochastic approximation, converges almost everywhere to the physical probabilities in a stationary environment. The linear reinforcement estimator has a nonzero variance even in the limit as $n \to \infty$, except when physical probabilities are 0 or 1 (see equation (A.9)). This allows it to track a nonstationary environment. Assuming the environment is stepwise stationary, it would be desirable to allow the estimates to converge to a set of values with a minimum amount of noise. Although the variance decreases with time (see (A.8)), faster convergence could be achieved by letting the learning rate α decay with time, with an annealing schedule similar to that used in simulated annealing [18]. On the other hand, once the variance is small, the estimator will not be able to respond very quickly to changes in probabilities, such as step changes or ramp changes of sufficient slope. Therefore, if

linear reinforcement is used with a decaying learning rate, some mechanism is needed to recognize when the estimates are grossly in error, so the learning rate can be reset to a higher value and resettled, allowing the estimates to reconverge.

To provide this function, PRIME uses a combination of short-term and long-term probability estimates, with sample mean performing the short-term estimation and linear reinforcement performing the long-term estimation. The long-term estimate is considered to be more accurate, so it is the value used in planning and in figuring the average effect probability and the support of a general rule.

The linear reinforcement estimator begins with a relatively high learning rate, which decays to a minimum learning rate and stays constant. The minimum rate is chosen to provide a small amount of adaptability, while keeping the variance below a desired threshold. Once this minimum is reached, the sample mean estimator starts running concurrently. After a number of samples are taken, the short-term estimate is compared to the long-term estimate. If the difference is greater than a certain threshold, the long-term estimate is initialized to the short-term estimate, and the learning rate is reset to some higher value and allowed to decay again. If the difference is not great enough to trigger a reset, the sample mean estimator starts again with new data, while the linear reinforcement estimator continues to fine-tune its estimates with the minimum learning rate. The number of steps for short-term estimation is chosen to provide a desired degree of accuracy in the short-term estimate, while keeping the estimation time to a minimum. The threshold for resetting the long-term estimation is derived from the accuracy of the short-term estimate.

Methods for selecting the constants used in the estimation procedures are given in the next two sections.

3.3.2.2. Short-term estimation parameters

The duration of the short-term estimation (T) affects the accuracy of the combined estimation procedure, as well as the speed with which the estimates respond to a change in physical probabilities. Since the estimate will be more accurate when the physical probability lies close to 0 or 1, it might be advantageous to reduce T when this situation is detected, perhaps by optimizing a loss function. Two methods [5, 11] are considered here which determine the optimal stopping time T of a sequential mean estimator, given a particular type of loss function. Both are asymptotically risk efficient in the sense of Starr [38].

Let $\hat{p}[n]$ be the estimate of the sample mean of $\{x[t]\}$ up to time n:

$$\hat{p}[n] = \frac{1}{n}\sum_{t=1}^{n} x[t]$$

The subscript i, indicating which effect this probability estimate pertains to, is suppressed for notational simplicity. Chow and Yu [5] propose a loss function of

$$L[n] = \sigma^{2\delta-2}(\hat{p}[n] - \mu)^2 + \lambda n \qquad (3.1)$$

where $\mu = E\{x\} = q$ (the physical probability being estimated) and $\sigma^2 = q(1-q)$ for the short-term estimation problem considered here, and $\delta,\lambda > 0$ are constants. This loss function places a cost on the variance of the data, the squared error of the estimate and the duration of the estimation. The objective is to find a stopping time which minimizes the risk:

$$R[n] = E\{L[n]\} = \frac{\sigma^{2\delta}}{n} + \lambda n$$

The optimal stopping time can be found if the variance σ^2 is known, but if it not known, it can be estimated by

$$V[n] = \frac{1}{n-1}\sum_{t=1}^{n}(x[t] - \hat{p}[n])^2$$

The authors show that a stopping rule of the form

$$T = \inf\{n \geq n_\lambda : n^{-1}V[n] + b[n] \leq \lambda^{1/\delta} a[n]\} \tag{3.2}$$

where $\lim_{n\to\infty} a[n]n^{-2/\delta} = 1$

$\lim_{n\to\infty} b[n] = 0^+$

$n_\lambda = O(\lambda^{-1/2})$

is asymptotically risk efficient; that is, as $\lambda \to 0$, the risk obtained from using (3.2) approaches the optimal risk if σ^2 were known.

To satisfy the conditions of (3.2), $a[n]$ and $b[n]$ are chosen to be:

$a[n] = n^{2/\delta}$

$b[n] = n^{-\beta}$, $\beta \geq 1$

The constant δ is chosen to be 2, to cause more estimation effort when the variance of the data is high. To find a reasonable value for λ, we can solve for λ in a typical case. If $\mu = q = 0.7$, we want the stopping time T to be around 20. Assuming that $V[T] \approx \sigma^2 = 0.21$, it follows that $\lambda = 9.15 \times 10^{-6}$. This leads to the requirement that the minimum stopping time be $n_\lambda = O(330)$, which is much too large. This is with $\beta = 1$, which allows the minimum n_λ. Knowing that n_λ increases with δ, we let $\delta = 1$, and all other parameters remain the same; the new result is $n_\lambda = O(81)$. It is not clear whether $20 = O(81)$, but with these choices of parameters, T ranges between 19 and 21, which is not a wide enough range to justify the extra computational effort in solving (3.2).

Ghosh and Mukhopadhyay [11] use a loss function similar to (3.1), but with $\delta = 1$. They show that the stopping rule

$$T = \inf\{n \geq 2 : n \geq \lambda^{-1/2}(V[n] + n^{-\gamma})\}, \quad \gamma \in (0, \tfrac{1}{4})$$

is asymptotically risk efficient. Again, a test case of $q = 0.7$ and $T = 20$ was used to find values of λ, and the result was that T ranged between 18 and 21 as q varied, and this was true for all values of γ in the domain.

In studying these two methods, the conclusion is that, for such a small sample size, the use of an optimization procedure to calculate the stopping time of the estimation does not result in enough savings to warrant the extra computation required; therefore, T should simply be chosen as a constant.

The selection of other parameters of the sample mean estimator is based on Tchebychev's inequality, which, stated in terms of sample mean estimation, is:

$$P\left\{ |\hat{p} - q| < \varepsilon \right\} \geq 1 - \frac{\sigma_x^2}{T\varepsilon^2} = \rho \tag{3.3}$$

where $\hat{p}$ is the short-term estimate of the physical probability q

ε is the upper bound in the estimation error

$\sigma_x^2 = q(1-q)$ is the variance of the Bernoulli random variable x

ρ is the lower bound of the probability of the estimation error being within ε

T is the number of samples in the short-term estimation.

The constants ρ and T are fixed at desired values, e.g., $\rho = 0.8$ and $T = 20$. Some experience with the procedure presented in this section will help determine what are reasonable choices, based on desired performance.

Once ρ and T are fixed, (3.3) will dictate ε as a function of q; that is, $\varepsilon = f(q)$. Then two functions can be defined:

$$B_l(\hat{p}) = q : q + \varepsilon(q) = \hat{p}$$

$$B_u(\hat{p}) = q : q - \varepsilon(q) = \hat{p}$$

These are the lower and upper bounds of the region where the correct value of the probability q lies, based on the short-term estimate $\hat{p}$ obtained after T steps, with probability $\geq \rho$. That is,

$$P\left\{ q \in [B_l(\hat{p}), B_u(\hat{p})] \mid \hat{p} \right\} \geq \rho$$

From these bound functions we can calculate the maximum estimation error:

$$\delta_{max} = \sup_{\hat{p}} \left(\delta(\hat{p}) : \delta(\hat{p}) = \max \left\{ |\hat{p} - B_l(\hat{p})|, |\hat{p} - B_u(\hat{p})| \right\} \right)$$

Using this quantity, we can say that

$$P\left\{ |\hat{p} - q| \leq \delta_{max} \right\} \geq \rho$$

This determines a triggering condition for resetting the long-term estimation: if $|\hat{p} - p| > \delta_{max}$, the long-term estimation is in error with probability $\geq \rho$, so it should be reset.

A low δ_{max} is desirable, since this will prevent the long-term estimate from straying too far from the actual probability. However, lowering δ_{max} requires either lowering ρ (the degree of certainty) or increasing T (the duration of the short-term estimation cycle). Lowering ρ too much will result in "false alarms," triggering a reset when it really is unnecessary. False alarms will always occur as long as the certainty is less than 1, but adjusting ρ allows some degree of control over their frequency. As mentioned before, increasing T increases the response time to changes in the environment.

3.3.2.3. Long-term estimation parameters

Selection of the minimum linear reinforcement learning rate α_{min} is based on minimizing the mean square error of the long-term estimate p. The mean square error is given by:

$$E\{(p[n] - q)^2\} = E\{p^2[n]\} - 2qE\{p[n]\} + q^2$$
$$= \sigma_p^2 + \mu_p^2 - 2q\mu_p + q^2$$
$$= \sigma_p^2 + (\mu_p - q)^2$$

Using the results of Appendix A, equations (A.4) and (A.8),

$$E\{(p[n] - q)^2\} = \alpha^2 q(1-q)\sum_{k=0}^{n-1}(1-\alpha)^{2k} + (1-\alpha)^{2n}(p[0] - q)^2$$

Defining $\delta = (p[0] - q)$ as the initial error, the mean square error function $MSE(\alpha,n,q,\delta)$ is given by:

$$MSE(\alpha,n,q,\delta) = \alpha^2 q(1-q)\sum_{k=0}^{n-1}(1-\alpha)^{2k} + (1-\alpha)^{2n}\delta^2$$

Given the *MSE* function and $n = T$ which is set in the design of the short-term estimator, the objective is to find α_m as a function of the initial error δ, which minimizes the mean square error over all q. One possibility is:

$$\alpha_m(\delta) = \alpha^* : \int_0^1 MSE(\alpha^*,T,q,\delta)dq = \min_\alpha \int_0^1 MSE(\alpha,T,q,\delta)dq \qquad (3.4)$$

The constant α_{min} is then determined by choosing δ_{min}:

$$\alpha_{min} = \alpha_m(\delta_{min})$$

We want α_{min} to be as small as possible, because this reduces the variance of p. Therefore, we choose δ_{min} to be the desired level of error when p converges to q, keeping in mind that, if the actual error δ is much larger than δ_{min}, then $\alpha_{min} < \alpha_m(\delta)$ and p will not converge very quickly.

The decay of α is exponential; the trajectory fits the form

$$\alpha[t] = \gamma^t$$

where $\gamma \in (0,1)$ is the decay rate. There are actually two trajectories for α. The first is called the initial settling; it is the decay from the very beginning of learning (the first

encounter of the condition and action in question). The second, called resettling, is the decay from the reset time, when the short-term estimator resets the long-term estimator. They are governed by two separate decay rates, γ_1 and γ_2 respectively. Since the final value of α is given (α_{min}), the decay rates can be found from the starting values and the desired times to reach the final value (settling times):

$$\gamma_1 = \log_{N_1}\left(\frac{\alpha_{min}}{\alpha_{max1}}\right)$$

$$\gamma_2 = \log_{N_2}\left(\frac{\alpha_{min}}{\alpha_{max2}}\right)$$

where N_1 is the number of steps of initial settling

N_2 is the number of steps of resettling

α_{max1} is the initial value of α for the initial settling

α_{max2} is the initial value of α for resettling.

When resettling begins, it is known that the reset probability estimate p is within δ_{max} of q, so $\alpha_{max2} = \alpha_m(\delta_{max})$. All other parameters in the list are arbitrary—N_1 and N_2 are chosen according to the desired performance, and $\alpha_{max1} \geq 0.3$ to reduce the effects of initial conditions and initial data.

3.4. Rule generalization

Generalization is the method of general rule induction in PRIME. There are actually two separate processes working independently: *condition-effect generalization*, which generates new rules with #'s in the condition and effect strings; and *action generalization*, which generates new rules with class names in the action sentence. These two methods do

not exhaust the possibilities of generalization in PRIME, but they represent the simplest and most easily implemented methods, and they are sufficient to illustrate the concept.

Generalization is meant to operate as a background process, building up the rule base when the machine is not occupied with planning or exploration, but it can also be called from within the exploration cycle or the plan execution cycle—this is the case in the current implementation.

With both types of generalization, the basic procedure is the same:

1. Generate a candidate general rule from two or more rules in the rule base; the source rules may be specific or general.
2. Gather all supporting rules.
3. Determine the support ratio and the effect probability of the candidate rule.
4. If the support ratio is great enough, add the candidate rule to the rule base.

As explained in §3.3.1, a supporting rule of a general rule is an instantiation of the general rule which exists in the rule base, and whose probability of effect exceeds a predefined support threshold. Also mentioned in that section was that the effect probability of a general rule is estimated as the average of the long-term probability estimates of the supporting rules. If the support threshold is high enough, these probabilities will be close enough that the average is a sufficient estimate for the general rule—recall that the general rule is only to be used in situations for which there is no specific rule in the rule base, so the more accurate estimate is always used when available.

The *support ratio* is defined as the ratio of *support* to *generality*. Support is the number of supporting rules for a given general rule, while generality is the number of possible instantiations of the general rule. The support ratio has two purposes: to decide whether to create a general rule (step 4), and to decide whether to delete a general rule. If the support ratio of a rule is higher than threshold for creating a general rule (the create

threshold), it is added to the rule base. If the support ratio is lower than the threshold for deleting a general rule (the delete threshold), the rule is removed from the rule base. The create threshold should be higher than the delete threshold; this provides some hysteresis, so small variations in the support do not unduly destroy the general rule.

As an example, suppose the candidate general rule is:

011#00#1001 robot1 pushto *box box* 110000#1011 ??

and the supporting rules in the rule base are:

01100011001 robot1 pushto box1 box2 11000011011 0.95

01110011001 robot1 pushto box1 box2 11000011011 0.90

01100001001 robot1 pushto box3 box1 11000001011 0.84

01100011001 robot1 pushto box2 box1 11000011011 0.97

The generality of the candidate rule is 36, the product of 3 possible direct objects, 3 possible indirect objects, and 4 possible condition-effect pairs. The support is, of course, 4, so the support ratio is 4/36 = 0.11, probably not enough to justify the candidate's insertion into the rule base. If it were inserted, however, its probability of effect would be the average of the support, or 0.915.

The next two sections describe the individual types of generalization in more detail.

3.4.1. Condition-effect generalization

Condition-effect generalization operates by selecting two compatible source rules from the rule base, and placing #'s in the condition and effect strings of the new general rule in positions where the two source rules differ. This amounts to a common symbolic induction method called "dropping conjuncts" [20].

Let the first source rule be represented by:

$$S1 = (C^{S1}, A^{S1}, E^{S1}, p^{S1}) \in \mathbf{R}_K = \mathbf{T}_K \times \mathbf{S} \times \mathbf{T}_K \times \Re$$

$$C^{S1} = (c_1^{S1}, c_2^{S1}, \ldots, c_K^{S1})$$

$$E^{S1} = (e_1^{S1}, e_2^{S1}, \ldots, e_K^{S1})$$

the second source rule by:

$$S2 = (C^{S2}, A^{S2}, E^{S2}, p^{S2}) \in \mathbf{R}_K$$

$$C^{S2} = (c_1^{S2}, c_2^{S2}, \ldots, c_K^{S2})$$

$$E^{S2} = (e_1^{S2}, e_2^{S2}, \ldots, e_K^{S2})$$

and the new general rule by:

$$G = (C^G, A^G, E^G, p^G) \in \mathbf{R}_K$$

$$C^G = (c_1^G, c_2^G, \ldots, c_K^G)$$

$$E^G = (e_1^G, e_2^G, \ldots, e_K^G)$$

In order to generalize $S1$ and $S2$ to form G, the effect probabilities of the two source rules must be above the minimum to be considered support, and the two rules must be compatible. The rules $S1$ and $S2$ are *compatible* with respect to condition-effect generalization if all of the following conditions hold for all $i \leq K$:

1. $A^{S1} = A^{S2}$
2. if $c_i^{S1} = \#$ then $(c_i^{S2} = \#$ and $e_i^{S1} = e_i^{S2})$
3. if $c_i^{S1} \neq \#$ then $c_i^{S2} \neq \#$
4. if $(c_i^{S1} \neq \#$ and $e_i^{S1} \neq e_i^{S2})$ then $(c_i^{S1} = e_i^{S1}$ and $c_i^{S2} = e_i^{S2})$

Conceptually, these conditions mean that the two rules *can* be generalized, i.e., there are no contradictions in the predicted effects. However, the conditions are stronger than that: they say that the condition strings must also be orthogonal, in the sense that there can be no

overlap between the sets of support rules for $S1$ and $S2$. Orthogonality guarantees that the support of the candidate rule formed by generalizing $S1$ and $S2$ has a support at least equal to the sum of their supports. This provides a minimum support test, placing a lower bound on the support ratio of the candidate rule which can be used to decide whether to add the new rule to the rule base, thereby postponing the search for the entire set of support rules until after such confirmation.

Once it has been determined that the rule G has satisfied the minimum support test, it is generated by the following algorithm, for all $i \leq K$:

if $c_i^{S1} == c_i^{S2}$ then

$c_i^G = c_i^{S1}$

$e_i^G = e_i^{S1}$

else

$c_i^G = \#$

if $(c_i^{S1} == e_i^{S1})$ and $(c_i^{S2} == e_i^{S2})$ then $e_i^G = \#$

else $e_i^G = e_i^{S1}$

end

The action of the new rule is the action of both source rules ($A^G = A^{S1} = A^{S2}$). If the new rule is found in the rule base, there is no need to add it again. Otherwise, the probability p^G is calculated by searching the rule base for all supporting rules, and taking the average of the effect probabilities of those rules. Now the new general rule is complete, and can be added to the rule base.

As an example, suppose the following two rules are stored in the rule base:

condition	action	effect	support	probability
0#110#1001	*lightrobot* gotoobj box2	1#11000001	5	.87
1#111#1001	*lightrobot* gotoobj box2	1#11100001	4	.95

Since these two rules are orthogonal, they can be generalized to form the candidate rule:

##10##1001 *lightrobot* gotoobj box2 1#11#00001

and the support of this rule is at least 9. Since the generality of the candidate is 32 (it can have 32 instantiations), the support ratio is at least 0.28. If the create threshold is 0.25, for instance, this rule would be added to the rule base, after the total support in the whole rule base and the average probability over these supporting rules are calculated.

It might appear that it is difficult to develop general rules, because of the amount of support required for a candidate general rule to be accepted. However, this amount of support, in the form of the support thresholds, can be decided by the designer, as he or she balances the need for highly general rules with the need for eliminating overgeneralized rules with little justification.

There is a real drawback to this method, however. There is no way to prevent the creation of general rules which already exist in the rule base; the rule base is searched after a candidate is generated, to see if the candidate is already present, but by then most of the work is finished. This results in much unnecessary labor, especially when the rule base is sufficiently rich—not only are there more pairs of rules to generalize, but it is more probable that a resulting candidate with sufficient support already exists as a rule in the rule base.

3.4.2. Action generalization

Action generalization operates by selecting two compatible source rules from the rule base which differ only in the actor or an object of the action sentence, and replacing the differing actors or objects with the class to which both belong. This is the same as another symbolic induction method called "generalization tree climbing" [20].

Let the source rules be represented by:

$$S1 = (C^{S1}, A^{S1}, E^{S1}, p^{S1}) \in \mathbf{R}_K$$

$$S2 = (C^{S2}, A^{S2}, E^{S2}, p^{S2}) \in \mathbf{R}_K$$

The rules $S1$ and $S2$ are *compatible* with respect to action generalization if all of the following conditions hold:

1. $C^{S1} = C^{S2}$
2. $E^{S1} = E^{S2}$
3. All corresponding components of A^{S1} and A^{S2} must be equal, except for a pair (O^{S1},O^{S2}) of actors or objects which differ.
4. O^{S1} and O^{S2} must belong to the same class, C_O.

Note that two compatible rules in this sense are necessarily orthogonal.

If $S1$ and $S2$ are compatible and their effect probabilities p^{S1} and p^{S2} are sufficiently high, then the candidate general rule is formed:

$$G = (C^{S1}, A^G, E^{S1}, p^G)$$

where A^G is A^{S1} with O^{S1} replaced by C_O. Next, the set of support rules for G is retrieved from the rule base, and if the support ratio of G is high enough, p^G is calculated from the average of probabilities of the supporting rules, and the new rule G is added to the rule base, if it is not already present.

Action generalization suffers the same drawback as condition generalization, for they both operate on the principle of generating candidate rules from pairs of existing rules.

Action generalization has an additional problem, though, a problem which may be called *class interaction*. This concept can be illustrated by showing the class interaction between the classes *lightrobot* and *box*.

The sentence robot1 climbonbox box2 has two objects (actually an actor and an object) which share some pertinent states, namely robot1_nextto_box2. In fact, every member of the class *lightrobot* and the class *box* share a similar state. This causes a state of the object to appear in the actor field of the condition string, and vice versa. If the actor is generalized, the sentence becomes *lightrobot* climbonbox box2, and the state robot1_nextto_box2, by virtue of its position in the actor field of the condition, becomes the state schema *lightrobot*_nextto_box2. When the rule is instantiated, an actor is chosen from the class *lightrobot*, and the mask string of that object is used to decode the state schema into the state robot1_nextto_box2 or robot2_nextto_box2. Likewise, the mask strings for the members of the class *box* are used to form and instantiate the state schema robot1_nextto_*box*. However, if both the actor and the object are generalized, there is no method of using the mask strings to form the state schema *lightrobot*_nextto_*box*, which can later be translated into the appropriate state when the action is instantiated. Therefore, specific boxes are always referred to in the actor field of the condition for *lightrobot* climbon *box*, and specific robots are always referred to in the object field of the condition. This makes it impossible to create a useful general rule for the action *lightrobot* climbon *box*, unless the values of these states are insignificant and #'s can be placed in the condition string, which is highly unlikely in most cases. Thus, class interactions make it difficult if not impossible to form rules with actions containing both class names. Class interaction causes this problem because the rule representation is not powerful enough to properly describe state schemata the way the systems of first-order logic and frames do.

3.5. Planning

3.4.1. Means-end analysis

The ultimate purpose of developing the rule base is to increase the machine's ability to plan effectively. Since the performance of PRIME will be evaluated primarily by the plans it generates, it is necessary to illustrate how a planner can use the rules induced by exploration and generalization.

Planning is the search for a sequence of specific actions which will transform the current environmental state into a state which satisfies a user-specified goal. A weak search method is needed for planning, because we cannot assume any *a priori* knowledge about the environment to guide the search. One such method, naturally suggested by the condition-effect structure of the rules in PRIME, is means-end analysis, which was used in GPS [8] and STRIPS [10]. Means-end analysis was described in §2.1.1.

Planning by means-end analysis can be transformed into a search through an and-or tree, and AO* (presented in [26]) could be used to search through the tree and generate a sequence of actions. Figure 3.3 illustrates a sample and-or tree for means-end analysis.

The nodes of the tree are ordered pairs containing one binary string (from **B**) and one trinary string (from **T**), which represent the environmental state and the goal to be achieved. The classifier matching procedure is used to decide whether an environmental state satisfies a goal. The root node contains the current environmental state and the user-specified goal, while the child nodes are subgoals of the planning process.

The and-links come in pairs, and each pair is associated with a specific action and a probability from a chosen rule. The action and the two child nodes denote the breakdown of the parent node into subgoals by applying the action. The first child node is composed of the state string from the parent node, and the condition of the rule for the action. The

second child node is composed of the effect of the rule for the action, and the goal from the parent node. Since the effect of the rule must be a binary state string in order to serve as the state for the second child, all #'s must be evaluated This is accomplished by first solving the first child node, so the condition is satisfied by a particular state string; then, knowing the values of the #'s in the condition, the #'s in the effect can be evaluated, and the second child node can be solved.

Each node may be broken down by applying a choice of rules, leading to a number of or-links composed of pairs of and-links. Referring to Figure 3.3, two rules are applied to the root node (S,G): the rules (C_1, A_1, E_1, p_1) and (C_2, A_2, E_2, p_2). Two rules are also applied to the node (E_1,C_4).

A node is solved if it is a leaf node, or if any pair of its children linked by and-links are solved. A leaf node is a node whose state satisfies its goal; the leaf nodes in Figure 3.3 are marked with implications, such as $E_1 \Rightarrow C_5$. If the root node is solved, the plan can be extracted by reading the actions from the solution tree from left to right. In Figure 3.3, the nodes of the solution tree are shaded, and the plan is A_3, A_1, A_6, A_8, A_4.

There are two elements which guide the search through the tree. First, the rules selected in order to solve a particular node (S_i, G_i) are chosen from the active rule set for the state S_i. This set is further refined by the means-end analysis procedure, which selects only rules which decrease the distance from the state to the goal; that is, for some applied rule $R = (C, A, E, p)$, $d(E,G_i) < d(S_i,G_i)$, for some distance function $d(s,g)$. For instance, this distance might be defined as the minimum number of bits in s which need to be flipped in order to satisfy g, something akin to a hamming distance.

The second guide to the search is a performance value associated with each node, which must be maximized in order to find an optimal plan. This value is the maximum probability of achieving the goal from the state, and is defined as the maximum probability

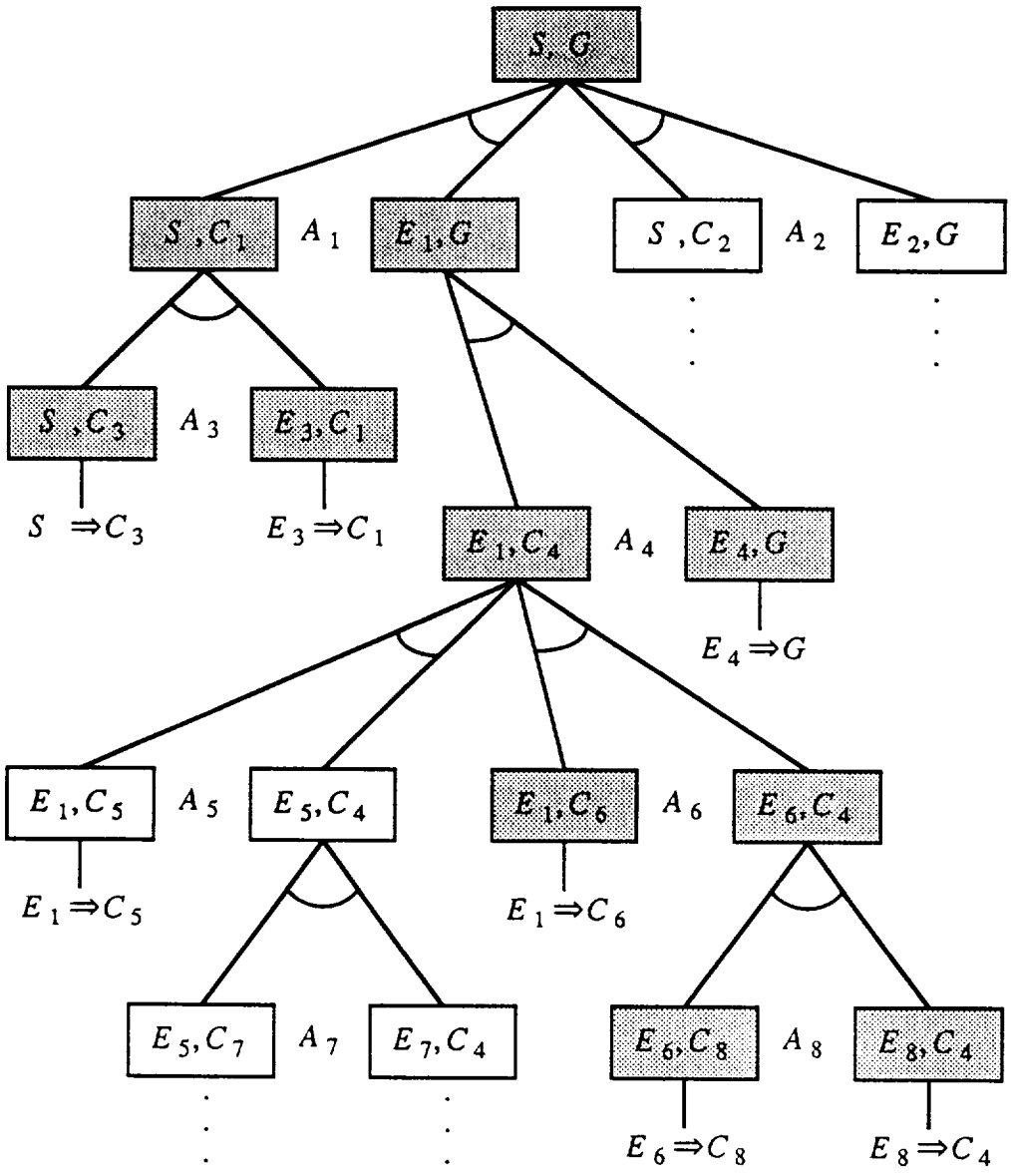

Figure 3.3 And-or tree

over all and-link pairs. The probability associated with each and-link pair is the product of the probabilities of the two children nodes, times the effect probability of the rule used to generate the two nodes. The probability for a leaf node is 1. Thus, the performance value of the root node, once solved, is the probability of success of the optimal plan, assuming

that the probability of success of each action is independent of every other action in the plan. In this type of search, a heuristic function may be used, which estimates the probability of a node which has not been expanded yet.

There are two problems with using means-end analysis in PRIME. The first, described in §2.1.1, is that means-end analysis does not guarantee that the optimal plan will be found, even if it could be derived from the existing rule base. This is primarily a problem with the function $d(s,g)$, which does not accurately determine how close state s is to goal g in the *solution space*.

The second problem concerns the default hierarchy in the rule base. Recall that, for any given environmental state, the most specific active rule for a particular action should be used, since it may serve as an exception to a more general rule—and even if it is not an exception, it still contains the most accurate estimate of the probability of effect. Also recall that if a general rule is applied which has #'s in its condition string, the values of these states remain unbound until they are decided by the effect of the previous action, i.e., when the first child node is solved. The default hierarchy problem arises when there is an exception to this general rule, whose activation is dependent on these unbound states. There is no way to tell whether the application of a general rule is valid at the time of application; the validity can only be checked when the unbound states are determined by previous actions, which are chosen later in the search. Solving this problem requires that every time an unbound state value is determined during the search, it should be propagated forward along all possible plans to check the validity of the general rules. This transforms an already complicated search procedure into an unwieldy one. Consequently, means-end analysis is not used in PRIME.

3.4.2. State graph search

It appears that the default hierarchy problem forces the planner to develop plans in the forward direction, i.e., from first executed action to last executed action, in order. This guarantees that the precise environmental state is known at each step in the planning process, so no guesswork is involved in deciding which rule to apply for a given action. The forward search procedure considered here uses a graph (called a *state graph*) to represent relations between states and actions, and the A* algorithm (proposed in [14] and presented in [26]) is used to conduct the search.

A state graph consists of nodes which represent particular environmental states, and links which represent specific action sentences. If two nodes are connected by a link, then there is a rule in the rule base which predicts that the execution of the action under the state indicated by the source node will result in the new state indicated by the destination node. Therefore, each node-link-node structure is an instantiation of a rule in the rule base. Each link also has a weight which is the probability of effect estimated by the rule. The initial node in the graph denotes the initial environmental state, and a final node is any node whose state satisfies the user-specified goal. Given these specifications, it is clear that a search through the rule base for an optimum plan to transform the environment from the initial state to a goal state corresponds to a search through a weighted state graph for an optimum path from the initial node to a final node, where optimum means "with maximum probability."

Figure 3.4 gives an example of a portion of a state graph. Listed are the rules which are active under the state indicated by the source node. These rules are instantiated and are used to form links to destination nodes, which denote the next state predicted by the rules. The probability beside each link is the effect probability of the corresponding rule, while the probability beside each node is the probability of reaching the goal from that node with

an optimum plan. The node probability is defined as the minimum, over all links, of the product of the link probability and the node probability of the destination node of that link. Thus, the probability of the parent node in Figure 3.4 is (0.9)(0.89) = 0.801. If the node in question is an unexpanded node (no links formed yet), a heuristic estimate of the probability of reaching the goal may be used.

Besides the effect probability which weights the link, there is also a constant weight assigned to each link, which effectively puts a cost on the length of a plan. This link weight takes the form of a probability, less than 1.0, which is multiplied by the effect probability of the rule when calculating the node probability. This weight is necessary because there are typically many rules in the rule base which have certain effects (i.e., their effect probabilities are 1.0), so there is no limit to the number of applications of these rules because there is no cost associated with using them. The link weight should be very close to 1 because longer plans with higher probability of success should still be preferred to shorter plans with lower probability of success. On the other hand, the weight should not be too close to 1, because it should help reduce the time of the search by limiting the length of a plan.

active rules under environmental state (0101110010101110110000101):

01011000101	robot1 pushto box2 *box*	01100100101	0.89
1##10011	robot2 climbonbox box1	11#10011	0.82
01011#01	*robot* gotobox box3	00110101	0.96
10111	robot3 gotoloc loc_b	10001	0.94

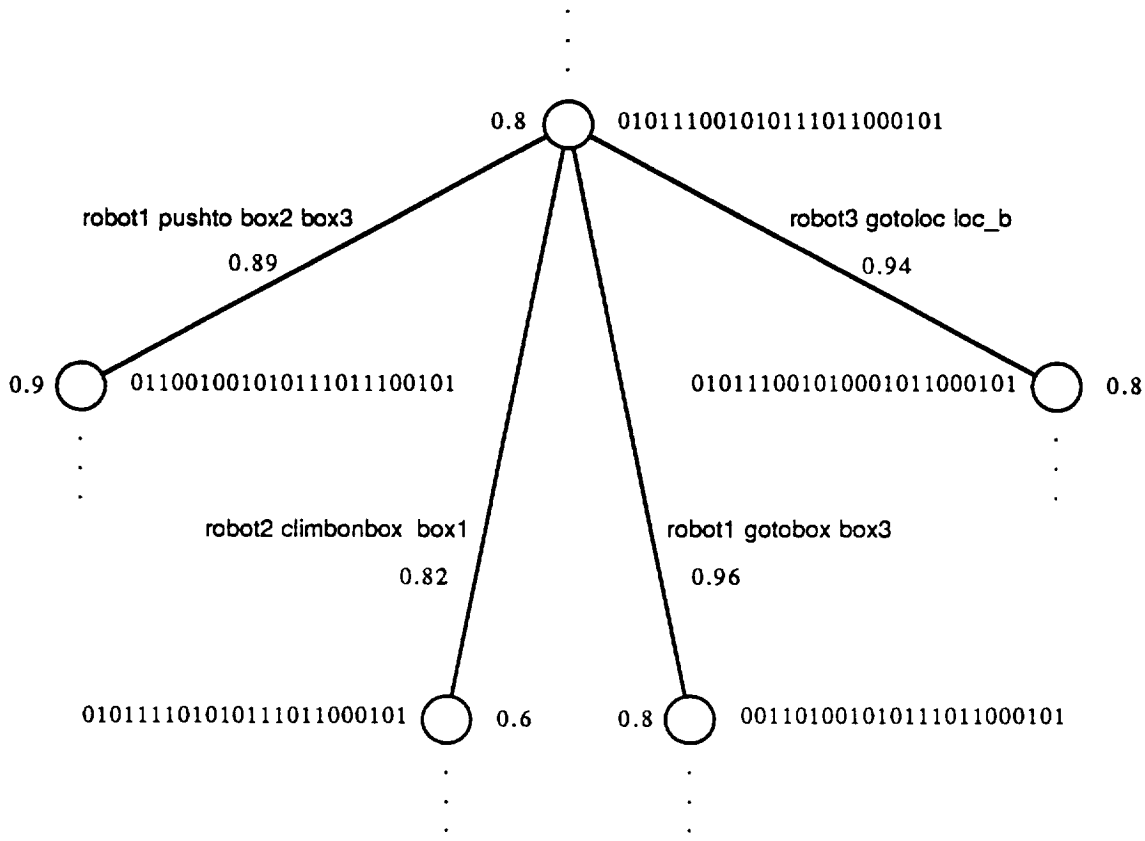

Figure 3.4 Partial state graph

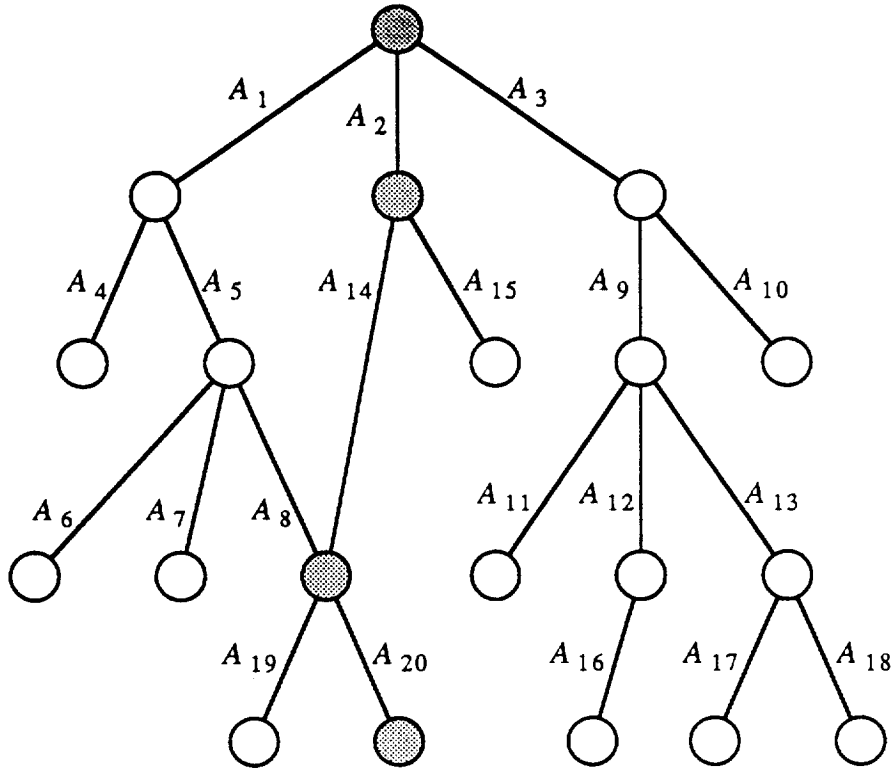

Figure 3.5 State graph

Figure 3.5 illustrates a sample state graph. Actually, this diagram represents a snapshot of a stage in the planning process; the subscripts of the actions indicate a possible order of generation of the links in the graph. Some of the leaf nodes are unexpanded nodes, i.e., nodes whose links and destination nodes have not been determined yet. The remaining leaf nodes are *terminal nodes*; these include nodes for which no rules are active, and nodes whose states satisfy the goal (a goal node). The bottommost shaded node is a goal node, so the path of shaded nodes represents a plan. The plan is compiled by taking the actions from the links along the plan path from top to bottom; in Figure 3.5, the plan is A_2, A_{14}, A_{20}. Notice that there may be more than one path from one node to another, but there is usually only one optimum path.

Although the full A* algorithm will not be presented here, a brief description of the search procedure follows:

1. Start with initial node, whose state is the current environmental state.
2. Expand current node.
3. Calculate probability of current node, given link and child node probabilities.
4. Update probabilities of all parent nodes.
5. If any child node is a goal node, return the path to the goal node as the plan.
6. If open list is empty, return failure.
7. Otherwise, remove maximum probability node from open list, and make it the current node.
8. Go to step 2.

The *expand node* subroutine in step 2 does the following:

1. Find active rules for state indicated by current node.
2. Keep only the most specific rules for each action, discarding all others.
3. If there is more than one rule of the minimum generality for any action, keep the one(s) with the most support, discarding the rest.
4. Eliminate rules which are not high enough in probability for planning.
5. Generate one link and destination node for each rule in the remaining set.
6. Estimate probability of reaching goal for each new node, using heuristic function.
7. Insert each new node into the open list.

Step 2 enforces the default hierarchy, favoring the least general rules available, while step 3 selects from among rules of equal generality the ones which are most supported by experience. Step 4 ensures that the effects of an action taken from any state are highly probable. These steps act as a filter which allows the planner to use the most accurate information available from the rule base.

The heuristic function in step 6 serves the same purpose as the distance function $d(s,g)$ in means-end analysis: to estimate the distance between the state and the goal. If the heuristic function is more optimistic than reality, then A* is *admissible*—it will always find an optimal plan if one exists. If the heuristic probability is always 1, then A* defaults to a blind, best-first search, which is admissible but very time consuming. A more appropriate heuristic function might be one which returns a decreasing probability with an increase in the hamming distance between the state of the node and the nearest goal state, reflecting the idea that more uncertainty is associated with an increase in the number of states to be changed. However, there is no guarantee that A* with such a heuristic will be admissible for an arbitrary rule base. If a better heuristic is needed, it should be tailored to the particular environment in which PRIME is expected to operate.

State graph search has several advantages over means-end analysis as a planning procedure. By always dealing with specific state values and avoiding unbound states, the state graph search method avoids the default hierarchy problem plaguing means-end analysis. Also, if the average number of active rules for a given condition is no greater than the average number of rules in the rule base producing a desired effect, then A* is preferable because the search tree would be narrower, and because A* is a simpler search procedure. Finally, because A* is admissible, it does not exhibit the incompleteness that means-end analysis does. If a particular version of A* does prove to be incomplete, then most likely the heuristic function in A* has the same difficulty which exists in the difference function $d(s,g)$ in means-end analysis: it is not accurately measuring the distance in the solution space.

3.6. Inclusion in Organization Level

To show that PRIME can operate in Saridis' Organization Level, we need to show that PRIME minimizes some entropy measure. In the Organization Level, this entropy represents the uncertainty of the success of any plan. Entropy is related to probability in the following way: Let $\Xi = \{\xi_i\}$ be a set of events, with an associated set of probabilities $\phi_i = P\{\xi_i\}$. Then the *entropy* of the partition is defined as

$$H(\Xi) = -\sum_i \phi_i \ln(\phi_i)$$

In the case of a plan, let

$$\Xi = \{\xi_0, \xi_1\}$$

$$\xi_0 = \{\text{plan succeeds}\}$$

$$\xi_1 = \{\text{plan fails}\}$$

$$\phi = P\{\xi_0\}$$

then it follows that

$$H(\Xi) = -\phi\ln(\phi) - (1-\phi)\ln(1-\phi)$$

Given the fact that the planner will always generate a plan with probability of success $\phi > 0.5$, it is clear that maximizing ϕ is equivalent to minimizing the entropy, and the A* planner presented in the previous section generates plans with maximum probability of success. In the current implementation, it is assumed that the event probabilities $\{p_i\}$ associated with each action in a plan are independent, so

$$\phi = \prod_i p_i$$

but this is not a requirement. If some other criteria were used to calculate the probability of success of a plan, then the method for calculating node probabilities in A* would be modified accordingly.

Of course, in planning it is actually the conditional probability of success of a plan which is maximized, conditioned on the current knowledge in the rule base. The role of the learning components of PRIME (specific and general rule induction) is to increase the accuracy and completeness of the rule base, thereby attempting to minimizing the unconditional entropy of any plan generated with these rules.

One issue remains, however. The action sentences and sequences of actions (plans) issued by PRIME are certainly suitable for input by the Coordination Level, but the Coordination Level may also feed back entropy values for each action, representing the complexity of performing that action. It would be useful to take these complexity measures into account when figuring the cost of a plan. One way to do this is to store the entropy feedback for an action in the rule which was used to generate that action. Then this complexity measure could be combined with the probability of effect when a node or link cost is evaluated. Of course, another heuristic function would be needed to estimate the complexity, but this could be assumed 0. If necessary, different methods for incorporating the feedback from the Coordination Level may be investigated in future research.

4. Implementation

This chapter describes the implementation of the various components of PRIME, whose general design was presented in §3. PRIME is written in C, so any references to specific data structures are expressed in C terminology.

4.1. Rule storage and recovery

Because the rule base in PRIME typically contains thousands of rules, these rules should be stored as efficiently as possible. In addition, the active set of rules for a given environmental state should be recalled quickly, since this is the primary operation with the rule base, especially in planning. This section details the implementation factors which lead to the achievement of these goals.

It was mentioned in §3.2.1 that the environmental state is represented as a binary string. This is actually stored in compressed form, taking one bit per character in the string. These bits can be accessed by name using a structure with one-bit fields, or they can be accessed as elements of an array using a *bitstring* structure, which contains a pointer to the binary state string and two indices used to quickly isolate a specific bit. The variable-length trinary string, which is used to store the condition and effect strings of a rule, is also stored in a compact form, with two bits per trinary digit (*trit*). Like the binary string, the elements of a trinary string can be accessed by using a *tritstring* structure, which contains two indices and a pointer to the compact trinary string. C facilitates the efficient storage and retrieval of individual bits and trits necessary for creating and activating rules, and this allows the conditions and effects of rules, the mask strings of objects, and the environmental state string to consume the minimum amount of memory possible.

The rule structure, in addition to the two compact trinary strings for condition and effect, also contain a structure of integers for the action sentence, a pointer to a statistics structure, a pointer to a general action sentence (if any), and a flag which indicates whether the rule is general or specific. A new rule structure is dynamically allocated whenever a new rule is added to the rule base, and pointers are used to access the rules. The statistics structures for general rules and specific rules differ in size and contents, so to minimize the storage requirements, these are dynamically allocated as well. As will be explained later, all instantiations of a general action rule are stored separately (for efficient activation), but all of these instantiations can point to the same general statistics structure, so duplication of information is minimized. The instantiations of a general action rule contain the specific actions in the action structure of the rules, but all of these instantiations have pointers to the same general action structure, which is also dynamically allocated; this also minimizes duplication. All of this is meant to demonstrate that the rules are stored as efficiently as possible, given the method by which the rules are activated.

Since rules are dynamically allocated, pointers are needed to recover them. These pointers are usually stored in linked lists called *rule lists*, which are also dynamically allocated. A global array of rule lists, indexed by the elements of the action sentence, provides a simple way of recovering all rules which have the same action. This array is useful for action generalization, but not for finding an active set of rules—for this, a modified trie structure is used, which keeps the search time linear in the length of the condition string.

A *radix search trie* [34] (or simply *trie*) is a type of tree structure in which the objects are pointed to by the nodes, instead of stored within the nodes. The objects are recalled by using a *key* which is composed of a string of digits. Each node in a radix search trie represents a decision about the value of a particular digit in the key. These digits may be

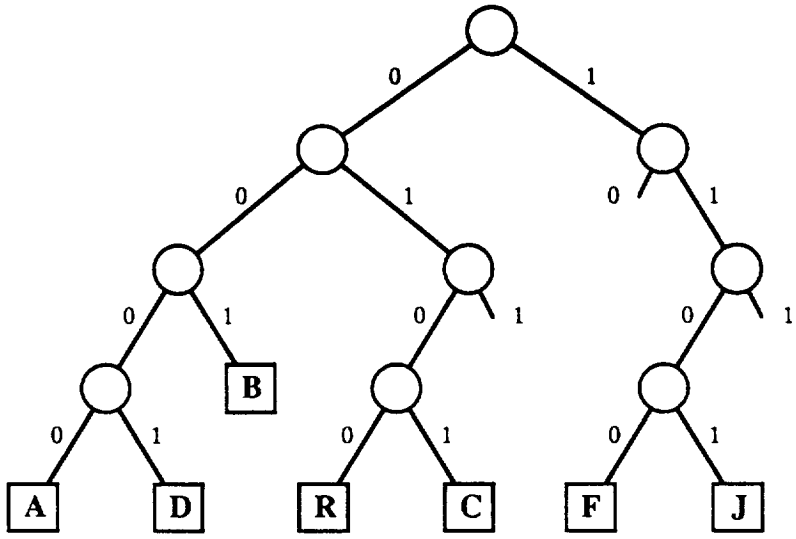

Figure 4.1 Binary search trie

binary or some other base (or *radix*)—Figure 4.1 gives a sample binary trie. To recall an object with a given key, start at the root of the trie, and take the branch indicated by the value of the first digit in the key; then take the branch indicated by the next digit. Continue branching until a branch dead-ends or an object is reached. If a dead-end is reached, the search has failed. If an object is reached, this is the only object which could match the key. A full comparison of the given key with the key of the object determines whether the search is successful. For example, in the trie illustrated in Figure 4.1, the object **R** can only be recalled with a key starting with 0100, while the object **B** can only be recalled with a key starting with 001. However, since the full key may not have been checked in this branching procedure, a full key comparison is necessary to verify whether the object reached is indeed the desired object, or whether there is no object with the given key. Thus, the search involves one key comparison and about $\log_2 N$ digit comparisons, for N objects with random keys.

A binary trie can be used to search for active rules, if the key is the binary state string and the objects are rule lists which indicate which rules are active under a certain

environmental state. Since a general rule is active under more than one state (by definition), it will be included in more than one rule list. In the case of general condition rules with specific actions, all of these lists will point to the same rule; no duplication of the rule is required. In the case of general action rules, though, it is necessary to specify which specific action is matched by the state string. For instance, if a general rule contains the general action robot1 climbonbox *box*, the environmental state may satisfy the conditions of the rule assuming the object is box1, but not if the object is box2. Therefore, separate rule structures must be stored for each instantiation of the action sentence, and although these structures will contain different specific actions, they will all point to the same general action and the same statistics structure. This separation of general action rules requires more storage, but it permits the automatic instantiation of the general action with the specific actors and objects whose conditions are matched.

Although the search time using a trie is small, it can be reduced even further by letting the branching complete the key comparison, obviating the full key comparison normally required at the end of the search. This can be accomplished by two different modifications of the normal trie. In the *full-depth trie*, an object is reached only after l branches, where l is the length of the key. Thus, the depth of the trie is constant, rather than varying with the particular keys used to store the objects. Examples of full-depth tries appear in Figures B.1 and B.2. Search using a full-depth trie proceeds in the same way as a normal trie, except that no final key comparison is necessary because, if the search should fail, it will fail by reaching a dead-end branch.

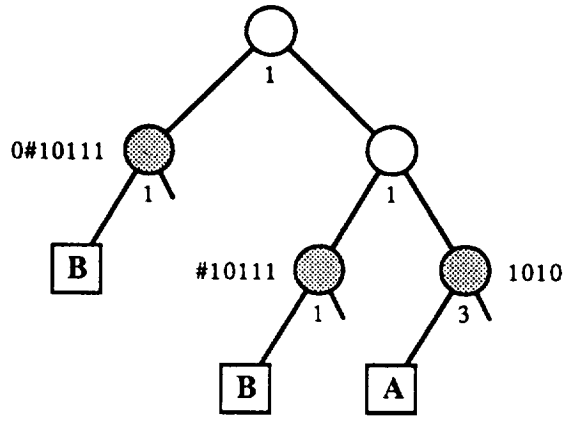

Figure 4.2 Compact trie

The second type of modified trie is the *compact trie.* An example of a compact trie is given in Figure 4.2. This trie is used to recall two objects, **A** and **B**, where

key of **A** = 111010##0

key of **B** = #0#101110

These keys may represent the condition strings of two rules, for instance. The compact trie is specially designed to handle keys with don't care symbols (#'s). Below each node is a number (a *shift count*), and beside each node may appear a trinary string (a *test string*). If a node is shaded, it is a *terminal* node. Searching using a compact trie proceeds as follows:

1. Start with a binary search key, such as a state string.
2. Make the current node the root node.
3. If there is a test string for the current node, test for a match between the test string and the leftmost portion of the search key (# matches 0 or 1). If there is a mismatch, the search failed; otherwise, discard the portion of the key which was tested, and proceed.
4. Shift the search key to the left by the number of bits indicated by the shift count of the current node, and test the last bit shifted out. If the last bit was a 0, select the left branch; otherwise, select the right branch.

5. If the selected branch points to nothing, the search failed.
6. If the current node is a terminal node, return the object (or list of objects) pointed to by the selected branch. Otherwise, make the node pointed to by the selected branch the current node, and go to step 3.

The compact trie is designed such that, when an object is reached, all of the bits of the search key have been shifted out, so all bits which need to be tested have been tested, and no extra key comparison is necessary.

A # in the key of an object may manifest itself in three ways: as a # in a test string, as a shift count greater than one, and as duplicate appearances of the object (actually duplicate pointers to the object). All three possibilities are illustrated in Figure 4.2. The duplicate appearance of **B**, for instance, occurs because if the first bit in the state string is a 1, either **A** or **B** could be selected, but if it is a 0, only **B** could be selected. However, if the first bit is a 1, the value of the second bit is enough to decide which object is the desired one, provided the rest of the bits match.

The search procedure for a compact trie might appear more complicated than that of the full-depth trie, but the elementary operations—bit shift and test—are identical. The test string of a node is actually stored as a list of signed shift counts: the search key is shifted left by the absolute value of the shift count, and if the shift count is positive, the last bit shifted out should be a 1; otherwise, it should be a 0. Using this notation, the test string

01##1#0

is stored as the list

-1, 1, 3, -2

This list, along with the shift count of the node itself, is used to implement the #'s in the keys of the objects; they skip over the corresponding bits in the search key, without testing them.

The representation scheme discussed above implies that the test string #0####1#####0#0####### takes the same space to store as the test string 1001. Since a constant-sized array is necessary to store the test string with each node, a ceiling must be placed on the number of non-# characters in the test string. It is desirable to keep this number small, to reduce the storage requirement of the trie, but if this value is chosen too small, some nodes must be split, resulting in more memory consumption. For example, Figure 4.3 illustrates the new compact trie after the maximum of four test bits per node is imposed. Since the test string of a node represents the common substring among the keys of all the objects which can be found under that node, this maximum should be chosen to reflect the average size of this commonality, which is probably quite small for a large rule base.

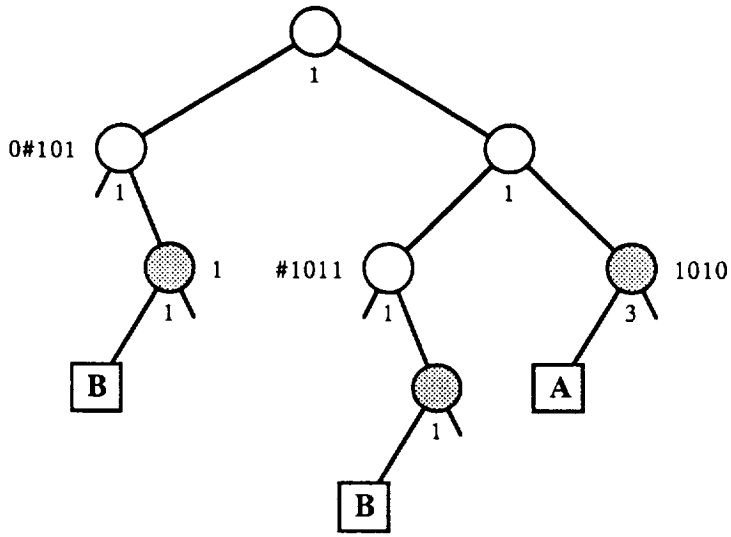

Figure 4.3 Compact trie with max testbits = 4

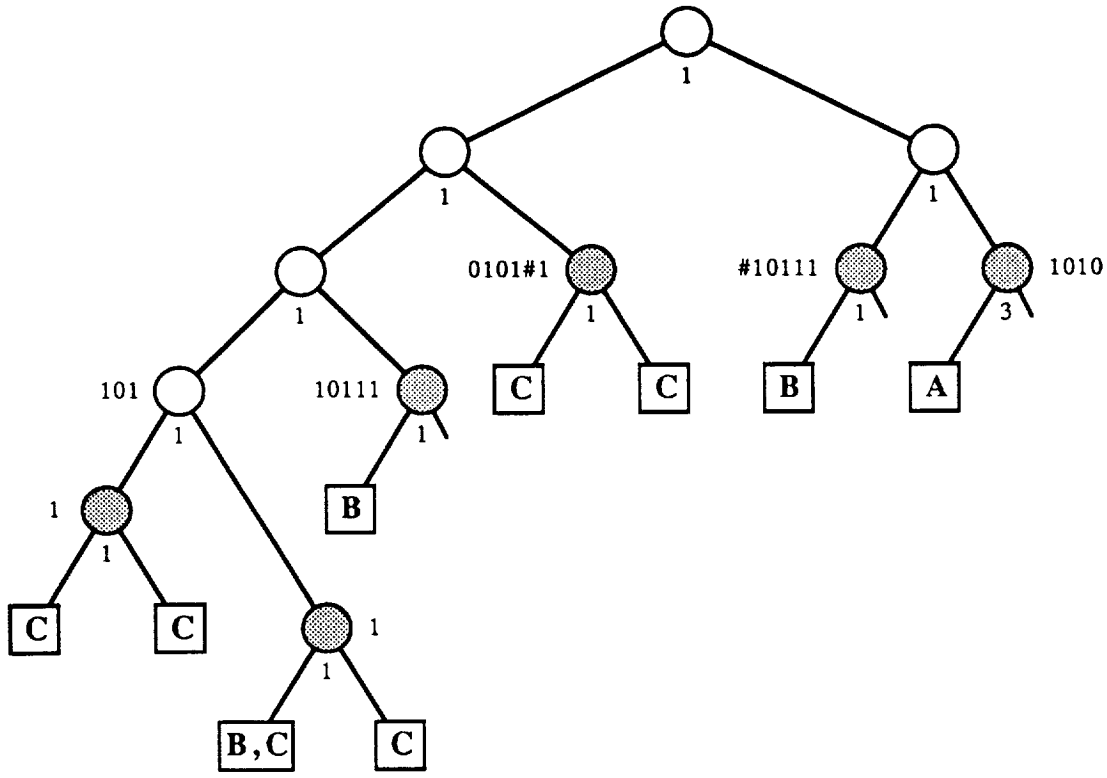

Figure 4.4 Compact trie with object C added

Figure 4.4 shows the compact trie when the object **C** is added, where

key of **C** = 0#0101#1#

The final # in the key forces **C** to appear in both branches of each terminal node which leads to it. Note that there is one state string (000101110) which matches the keys of both **B** and **C**; thus, **B** and **C** appear together at the place in the trie where this state string would lead.

One might object to the frequent duplication of pointers to objects in the trie in Figure 4.4, but this redundancy is no less extensive in the normal trie or the full-depth trie. This is because the compact trie offers two additional methods of accounting for #'s, test strings and shift counts, which are not available in the other two trie structures. As an example, the full-depth trie would require a "box" for each binary search key which would match the

key of an object—thus, 4 boxes for **A**, 4 boxes for **B**, and 8 boxes for **C**. Meanwhile, as Figure 4.4 demonstrates, only 1 box is required for **A**, 3 boxes for **B**, and 6 boxes for **C**.

If the objects represent rules and the boxes are rule lists, then the compact trie stores rules with equal or fewer rule lists than the full-depth trie. Even assuming that the number of boxes are equal, the compact trie is still more memory efficient than the full-depth trie, even though the sizes of the nodes themselves are larger for the compact trie. The node of the full-depth trie consists only of two pointers, which take 8 bytes total. The node of the compact trie consists of two pointers, 6 chars for the six test bits of the test string, a char for the shift count, and a char for the terminal flag, totalling 16 bytes. Changing the terminology slightly, let the objects be the boxes, or rule lists, and let the number of objects be $N = 10{,}000$ and the length of the key be $m = 30$, corresponding to a rule base of about 2,000 rules and a state string of 30 states. Using these figures and equations (B.1) and (B.2) developed in Appendix B, the estimates of the number of nodes in the full-depth trie are:

full-depth, best-case: 16.4K nodes

full-depth, worst-case: 170K nodes

By Appendix C, on the other hand, the estimates of the number of nodes in the compact trie are:

compact, best-case: 10K nodes

compact, worst-case: 20K nodes

This leads to the following estimates of storage requirements:

compact, best-case: 160K bytes

full-depth, best-case: 131K bytes

compact, worst-case: 320K bytes

full-depth, worst-case: 1360K bytes

These figures clearly indicate that the compact trie is indeed more compact. In addition, there is virtually no performance cost in using the compact trie, since the operations used to traverse the compact trie are the same as those used to traverse the full-depth trie.

Until now, is was assumed that all the rules in the rule base would be accessed by one trie. The set of active rules for a given environmental state would be recalled by traversing the trie using the full binary state string and returning the rule list encountered at the end of the search, if any. This implies that the key of each rule is the same length as the state string, and would be formed from the condition of the rule by "expanding" the condition of the rule to form a trinary state string, by reversing the process of condition compilation using the actor and object masks, and placing #'s in the unspecified positions. This can certainly be done, but it turns out that it is more memory efficient if a separate trie is maintained for each specific action, and all rules within a given trie contain the same action sentence. By using this method, the key of each rule can simply be its condition string; this can shorten the depth of the tries considerably. To search such a trie, the state string must first be compiled into a key the size of the condition string for the particular action. Of course, instead of searching one trie, we must now search one trie for every possible action sentence and take the union of the results. However, the number of syntactically correct action sentences is small relative to the number of possible action sentences, and the search in each nonempty trie is likely to be much shorter than the search through a common trie. Therefore, the memory savings costs very little in total search time, and in fact, insertion of new rules is much quicker. PRIME currently implements this multiple compact trie arrangement for searching for active rules.

4.2. Probability estimation

To choose the parameters of the short-term and long-term probability estimators, the procedures developed in §3.3.2.2 and §3.3.2.3 are followed. The duration of the sample-mean estimator is selected as $T = 20$; later we will see how shortening the duration to $T = 10$ affects the performance. The short-term estimation error is desired to be within $\varepsilon = 0.2$ with a minimum probability of $\rho = 0.8$. These figures allow the construction of Table 4.1 which indicates values of the lower and upper bounds of q, and the maximum error in the estimate $\hat{p}$.

Table 4.1 Errors in sample-mean for T=20, ρ=0.8

$\hat{p}$	$B_l(\hat{p})$	$B_u(\hat{p})$	max error
0.1	0.024	0.336	0.236
0.3	0.131	0.549	0.249
0.5	0.276	0.724	0.224

Table 4.1 shows that $\delta_{max} \approx 0.25$. This implies that the long-term estimate p should be reset when $|p - \hat{p}| > 0.25$.

The constant δ_{min} is selected to reflect the desired error in the long-term estimation; in this implementation, $\delta_{min} = 0.05$. By equation (3.4), $\alpha_{min} = \alpha_m(\delta_{min}) = 0.015$. Figure 4.5 shows graphically that $\alpha = 0.015$ minimizes the overall mean square error of p with $\delta = 0.05$, considering all values of q. Also using equation (3.4), $\alpha_{max2} = \alpha_m(\delta_{max}) = 0.1$, which is verified by Figure 4.6.

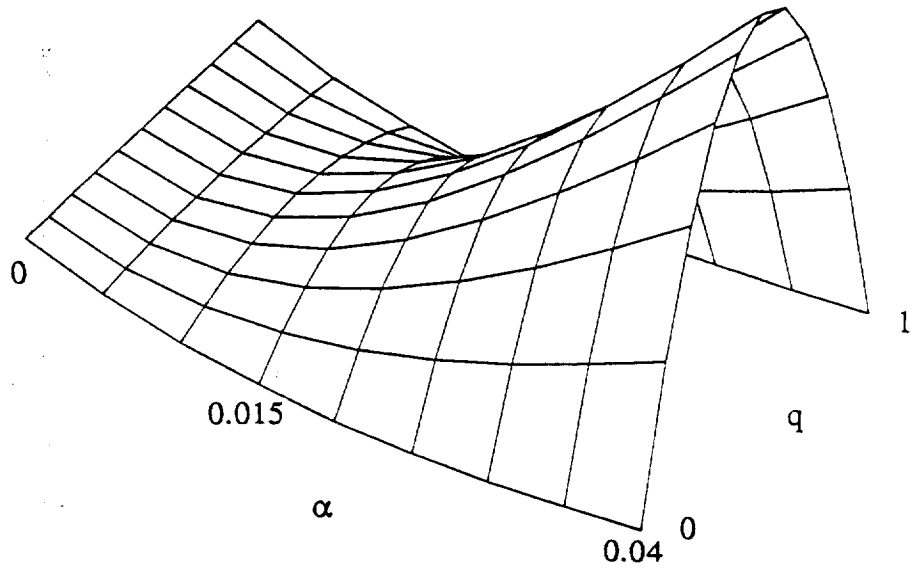

Figure 4.5 Mean square error, $\delta = 0.05$

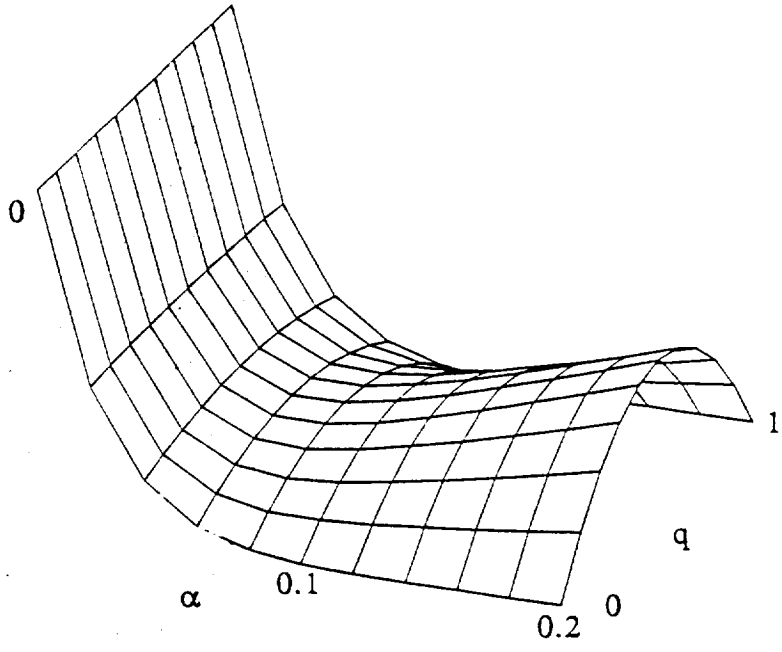

Figure 4.6 Mean square error, $\delta = 0.25$

For the remaining parameters, some experimentation shows that the following choices yield satisfactory results:

$$\alpha_{max1} = 0.3,\ N_1 = 30,\ N_2 = 20$$

This leads to the decay rates of

$$\gamma_1 = 0.9050,\ \gamma_2 = 0.9095$$

The graphs in Figures 4.7-4.10 illustrate the trajectories of three long-term effect probability estimates for the same action (which must sum to 1.0), while trying to track ramp changes and step changes in the physical probabilities. The large jumps in the estimates indicate when the estimates are reset, due to a difference between the long-term and short-term estimates which is greater than δ_{max}. The α constants can, of course, be changed if a different balance is desired between tracking ability and noise level of the estimates.

Figure 4.10 shows a delay of about 30 steps between the step changes in q and the resetting of p. (The fact that the delay is longer than T is caused by the timing of the change.) If this delay is considered too long, T could be decreased to 10. Letting $\rho = 0.75$, Table 4.2 gives the maximum errors in the short-term estimate.

Table 4.2 Errors in sample-mean for T=10, ρ=0.75

$\hat{p}$	$B_l(\hat{p})$	$B_u(\hat{p})$	max error
0.05	0.005	0.352	0.302
0.1	0.017	0.411	0.311
0.3	0.106	0.609	0.309
0.5	0.233	0.767	0.267

Table 4.2 indicates that $\delta_{max} \approx 0.3$; all other constants remain the same. The results of changing T and δ_{max} are shown in Figures 4.11 and 4.12, which use the same random data as Figures 4.9 and 4.10. The tracking of ramps does not improve, but the response to the step function is much quicker.

The cost of decreasing T is either an increase in δ_{max} or a decrease in ρ. In this example, $\delta_{max} = 0.3$ corresponds to $0.7 < \rho < 0.75$. Thus, the criterion for resetting is greater than the original value, and the certainty that resetting is necessary is smaller. The decreased certainty, in particular, leads to an increase in the frequency of unnecessary resets in the estimates, or "false alarms." The effect of false alarms when $T = 10$ and $\delta_{max} = 0.3$ can be seen in Figures 4.13 and 4.14, where most of the large jumps in the probability estimates appear unwarranted. This increase in false alarms justifies the decision to retain the original values of $T = 20$ and $\delta_{max} = 0.25$.

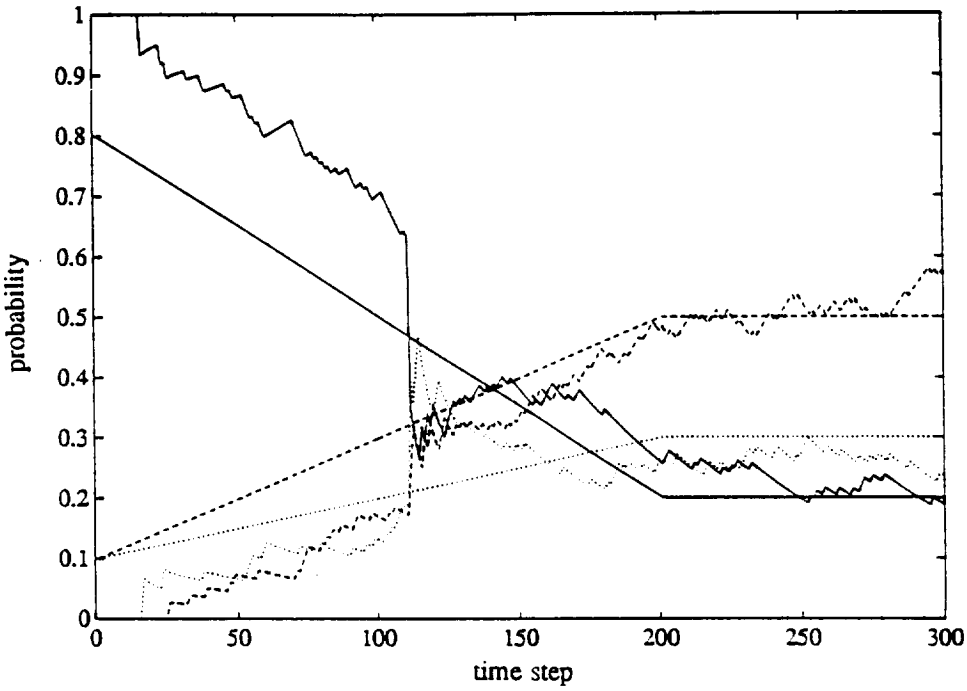

Figure 4.7 Estimation ramp response, $\delta_{max} = 0.25$, sample 1

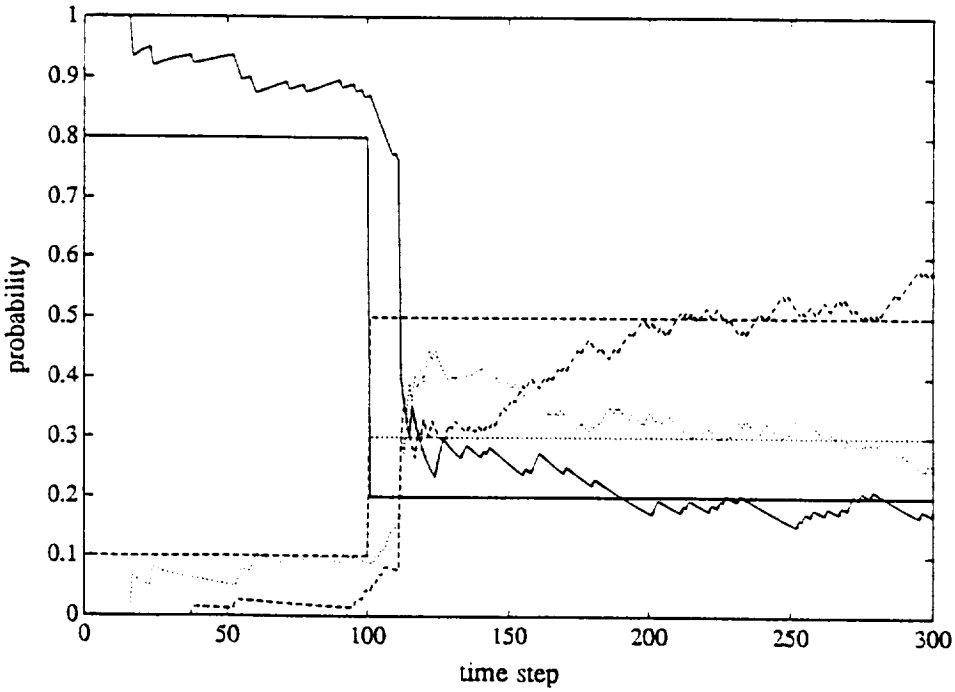

Figure 4.8 Estimation step response, $\delta_{max} = 0.25$, sample 1

C-2

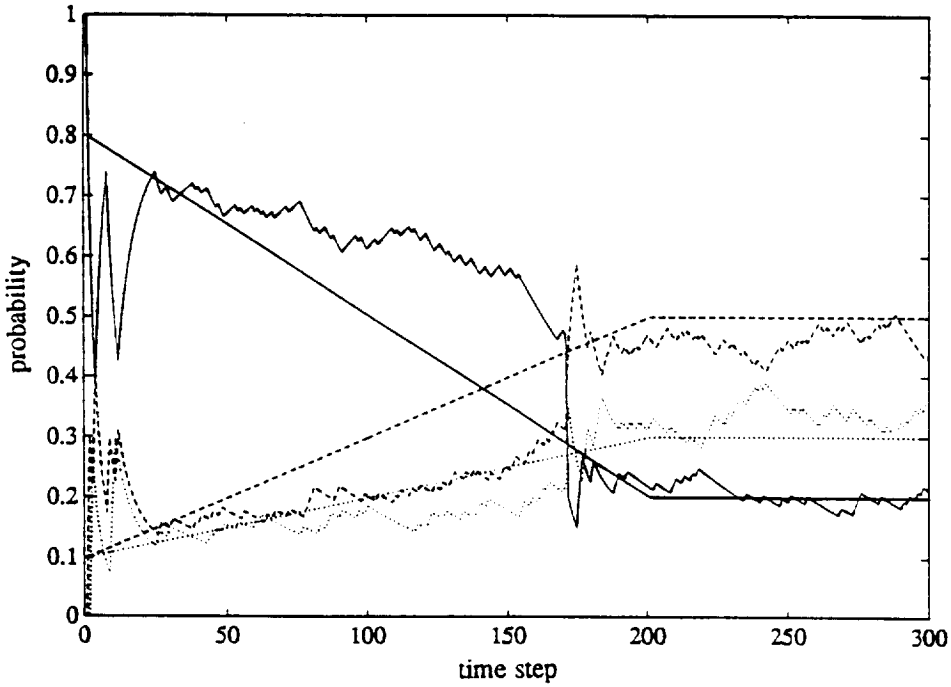

Figure 4.9 Estimation ramp response, $\delta_{max} = 0.25$, sample 2

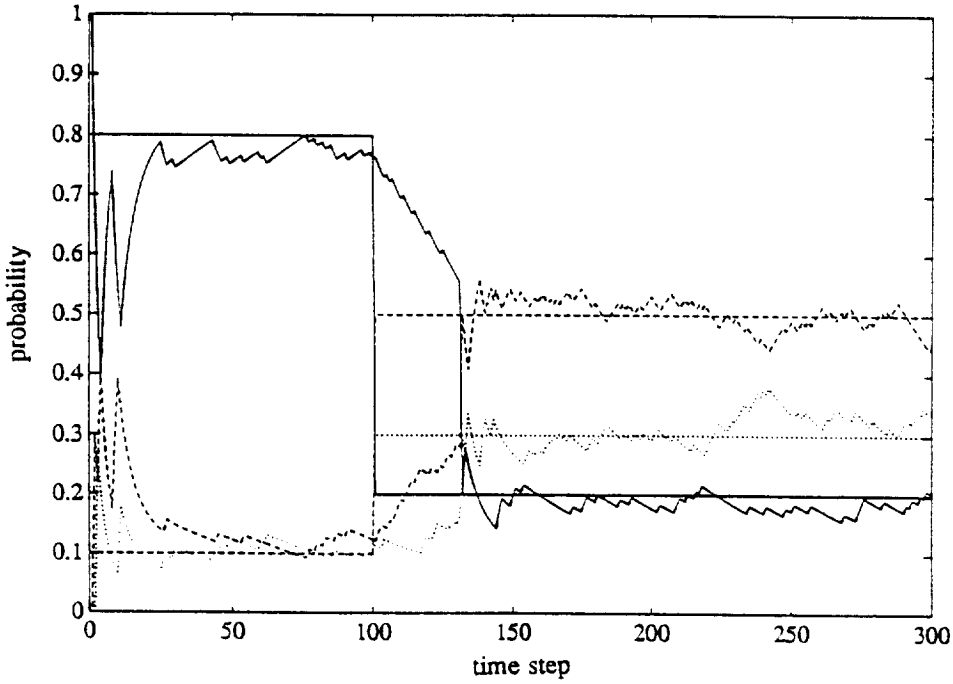

Figure 4.10 Estimation step response, $\delta_{max} = 0.25$, sample 2

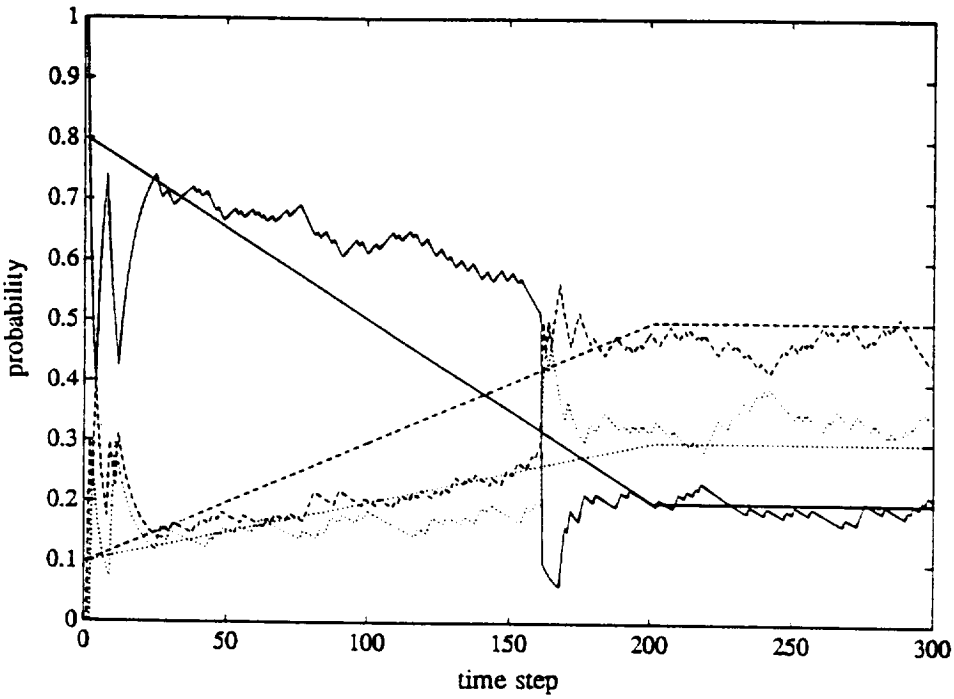

Figure 4.11 Estimation ramp response, $\delta_{max} = 0.3$, sample 1

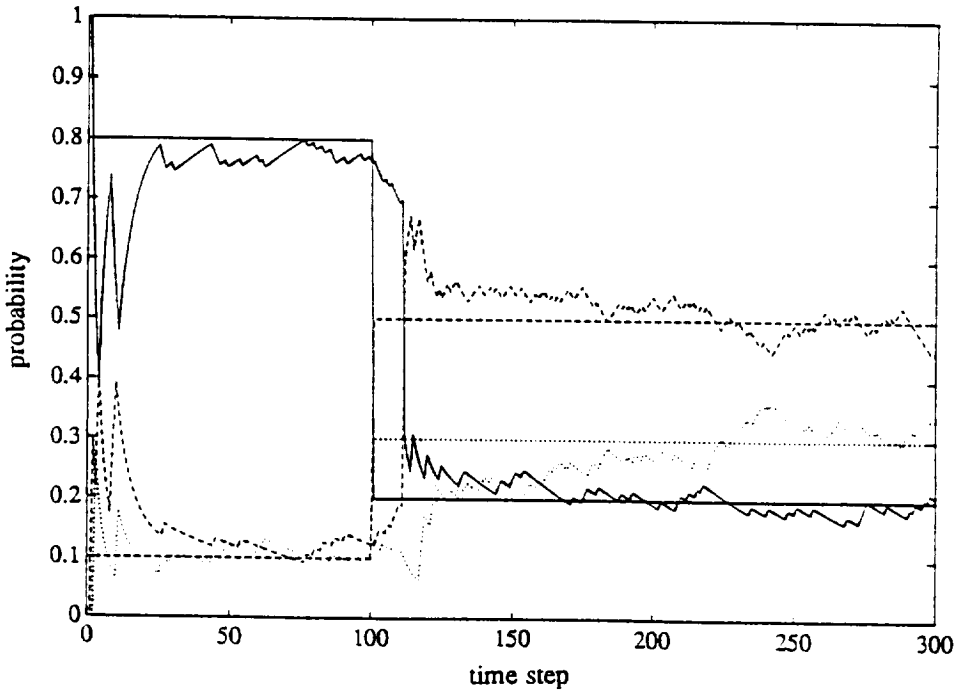

Figure 4.12 Estimation step response, $\delta_{max} = 0.3$, sample 1

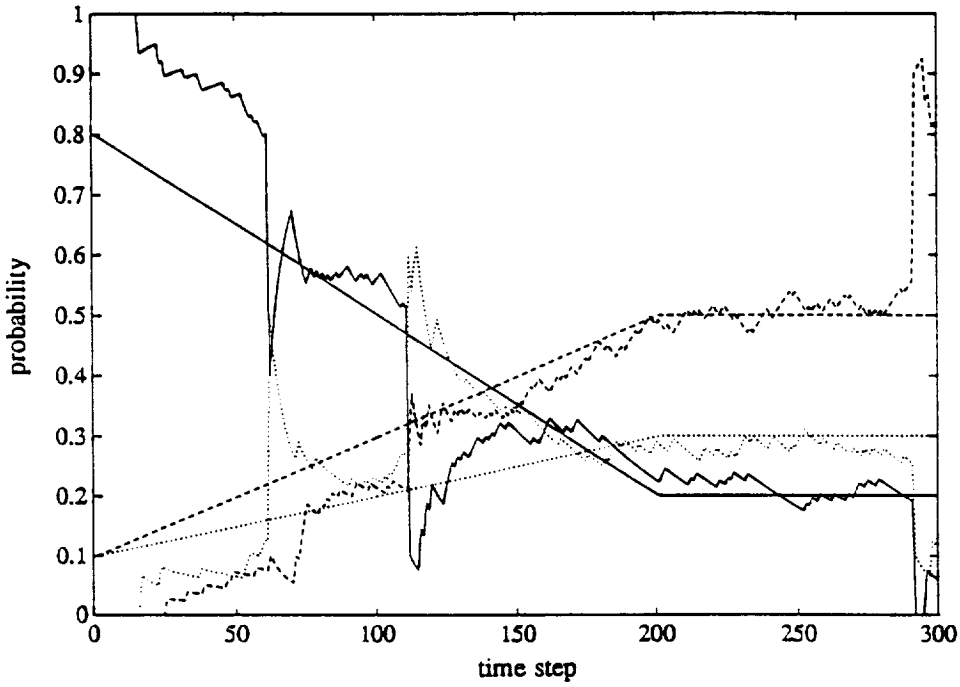

Figure 4.13 Estimation ramp response, $\delta_{max} = 0.3$, sample 2

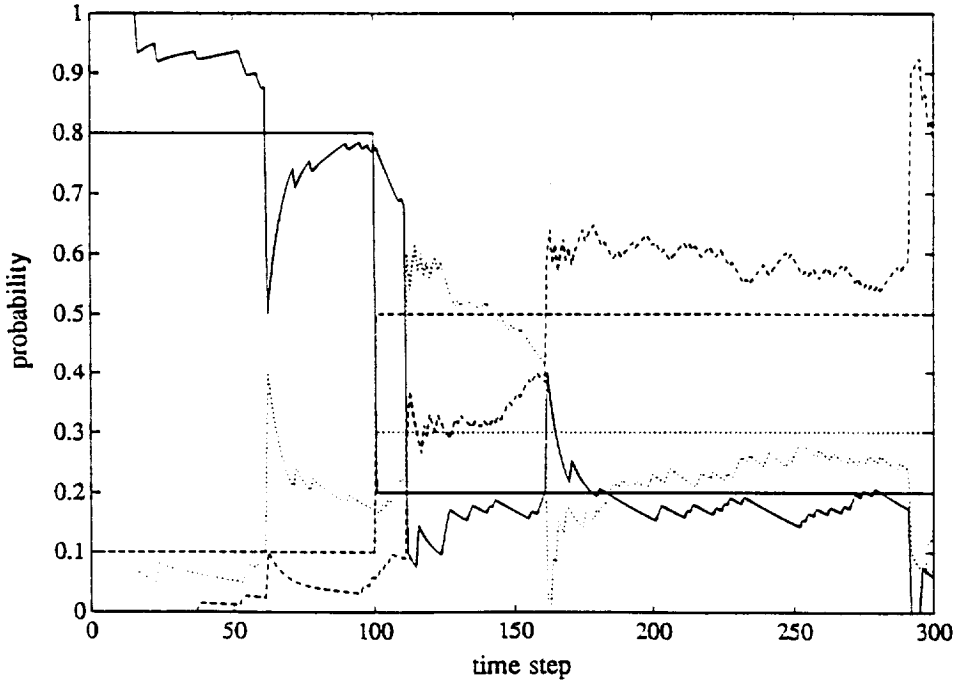

Figure 4.14 Estimation step response, $\delta_{max} = 0.3$, sample 2

4.3. Generalization

Generalization was described in §3.4 as a background process, but at this point it is implemented as a procedure which is called every time a rule is updated. Because generalization is executed serially with the other functions of PRIME, its run time must be kept short. Therefore, only a small set of rules are generalized at a time: the set of rules which were active under the previous environmental state, and which are concerned with the action which was just executed. This tends to generalize to a greater degree the rules which are more frequently active. In one generalization cycle, candidate general rules are created by applying the generalization algorithms of §3.4.1 and §3.4.2 to each pair of rules in the selected set, and only those candidates with a high enough support ratio are added to the rule base. With this limitation on the rules to be generalized, the generalization procedure accounts for a relatively small portion of the CPU time used in the exploration cycle, but this portion grows as the number of active rules grows.

There are two constants which apply to the insertion and deletion of general rules from the rule base: the creation support threshold and the deletion support threshold. These are support ratio thresholds which determine how much support is required to create a general rule, and how much support is required to keep a general rule from being deleted. In the current implementation, these thresholds are 0.25 and 0.125, respectively.

Another threshold constant was added, α_{gen}, which limits the proportion of general rules to specific rules. An active rule with the specified action is not considered for generalization unless its learning rate α is below α_{gen}. The effect is that the machine must have some experience with a specific rule before trying to generalize it. This reduces the proliferation of dubious general rules based on infrequent occurrences, and it allows this space to be used by more specific rules. Of course, if α_{gen} is too low, then the population of general rules will be so low that they will not fill the gaps of knowledge which are not

covered by the experience of the machine. In the present implementation, $\alpha_{gen} = 0.25$, effectively requiring three exposures to the specific rule before generalizing, given that the initial decay rate is $\gamma_1 = 0.9050$.

4.4. Planning

The state graph search described in §3.5.2 is the planning method used in PRIME. The two main data structures used in planning are the node and link types for the planning tree. The node type contains the (compressed) state string for the node, the probability of that state occurring, the probability of reaching a goal from the start state using the path passing through this node, a pointer to the previous link which leads into this node, and a number of pointers to next links which lead to lower levels in the tree. The maximum number of next links is predefined; this value controls the width of the planning tree. The link type contains a pointer to the source node, a pointer to the destination node, and a pointer to the rule used to predict the effect of the action which this link represents.

There are a few other data types used during a planning session. In accordance with the standard A* algorithm, lists of open nodes and closed nodes are maintained—open nodes are new nodes which have not been expanded yet, and closed nodes are the expanded nodes. However, these node lists are actually stored as tries, for quick access by state string value; when a new node is created, all open and closed nodes must be searched, to determine if the new state has been generated before. Nevertheless, the open nodes are also stored in a linked list which is ordered by increasing cost, allowing quick access to the cheapest open node (or the one with the highest probability of reaching the goal) for expansion. In addition to these structures, the planner maintains a list of terminal nodes—nodes which have no branches emanating from them. If planning fails, these terminal

nodes can be searched to find the partial plan, ending with one of these nodes, which appears most likely to lead toward the goal. Needless to say, all of these data types are dynamically allocated, and freed when planning is complete.

The open and closed node tries are full-depth tries, because the keys used to store the nodes are binary—they do not contain any #'s. The radix of the trie is determined by its memory efficiency. A worst-case trie structure is assumed, because the best-case structure occurs only when the last bits in the state string are the ones which differentiate the states of the planning nodes; it is much more likely that the states of the planning nodes can differ in any random state in the string, resulting in a full, open tree such as the one in Figure B.1. Using equation (B.1) and the fact that each trie node requires $4p$ bytes, the estimates of the storage requirements for various values of n and p are listed below:

$n = 1, p = 2$: 6,262 bytes

$n = 4, p = 16$: 13,966 bytes

$n = 8, p = 256$: 127,089 bytes

From these figures it is evident that the binary full-depth trie is the best choice, despite the speed gained by testing larger chunks of the state string.

There are four factors which can potentially limit the search in planning. The first is the aforementioned limit of the number links originating from an expanded node; this is currently set at 15, and this value has proven high enough to have no effect on planning with the rule bases used in simulation so far. The choice of this limit should depend on the maximum number of actions which can be executed in *any* state, since links represent specific actions.

The second factor is the minimum probability of a plan leading to the goal; this constant is used to discard a node which appears to be too far from the goal, or which is

part of a path which is so long that the probability of success is significantly diminished. This probability is calculated by

$$p(s_0,g) = p(s_0,s_i)\cdot h(s_i,g)$$

where $p(s_1,s_2)$ is the probability of reaching the state s_2 from the state s_1 (given by the product of the link probabilities along the path from s_1 to s_2); $h(s,g)$ is the heuristic estimate of the probability of reaching the goal g from the state s; s_0 is the initial state; s_i is the state of the current node (i); and g is the desired goal. If the value of $p(s_0,g)$ calculated using node i is smaller than the threshold p_{min}, then node i is discarded. This threshold prevents the planner from exploring plans which do not appear to be lucrative, and it also allows the planner to eventually decide it has failed, instead of continuing a search perhaps indefinitely. The value of this threshold is set at 0.75 in the current implementation of PRIME.

The third factor is the heuristic probability estimation function $h(s,g)$. The use of this function allows the planner to prioritize the possible search directions at every stage in the planning process. If $h(s,g)$ is more optimistic than reality, then the search is admissible, and the planner will always find the optimal plan if one exists. The heuristic function currently used in PRIME is

$$h(s,g) = \frac{N - d(s,g)}{N}$$

where $d(s,g)$ is the hamming distance between the state s and the nearest state which satisfies the goal g. This function decays linearly with the distance between the state and the goal, which effectively penalizes states which are far from the goal while still remaining rather optimistic about reaching it. There is no reason to believe that this heuristic is more optimistic than reality for an arbitrary environment, so there is no guarantee that it results in an admissible search procedure, but it has proven satisfactory in simulation. A more

accurate heuristic function would have to be tailored to the particular environment in which PRIME would operate.

The fourth factor is the link weight, w_l, which effectively puts a cost on the length of a plan. Recall that this weight is should be a value less than but very close to 1.0; the chosen value is 0.98.

Once a plan is generated, each action in the plan is executed, the effects of these actions are observed, and the rule base is updated accordingly. It may be the case that some of the actions in the plan may sometimes lead to no change in state at all; that is, there may be a small probability that an action in the plan is ineffective. If this occurs, the action is retried until either the action does produce some change in the environment, or the retry count passes a predefined threshold, such as 10 tries. The plan has failed if an action is ineffective after these retries, or if the whole plan is executed but the goal is still not satisfied. One could elect to replan from this point, or explore for a while to develop further knowledge, but no such recovery mechanism is currently implemented in PRIME.

4.5. Environment simulator

The environment model chosen to test the operation of PRIME is based on a subset of NASA's Flight Telerobotic Servicer Task Analysis Methodology [25], which provides a language and methodology for specifying telerobotic operations, especially in the assembly and maintenance of Space Station Freedom. PRIME is used to issue commands at the Telerobot Task Level; tasks in this level are performed by a single work system (or actor) on a single object. The definitions of the acronyms used in this section are given in Appendix D.

The sets of actors, verbs and objects are:

A = {SSRMS, RSM, FTS1, FTS2, MT1, MT2}

V = {transport, position, remove, replace}

U = {FTS1, FTS2, IEA1, IEA2, ORU1, ORU2, ORU3, ORU4,
MSC1, MSC2}

The actor classes are:

$\mathbf{C}_A$ = {*positioner, FTS, MT*}

positioner = {SSRMS, RSM}

FTS = {FTS1, FTS2}

MT = {MT1, MT2}

The object classes are:

$\mathbf{C}_U$ = {*FTS, ORU, MSC, IEA, null*}

FTS = {FTS1, FTS2}

ORU = {ORU1, ORU2, ORU3, ORU4}

MSC = {MSC1, MSC2}

IEA = {IEA1, IEA2}

null = ∅

There are 44 states in the state set, as follows:

Z = {MSC1_near_IEA1, MSC1_near_IEA2,
MSC2_near_IEA1, MSC2_near_IEA2,
FTS1_at_IEA1, FTS1_at_IEA2,
FTS1_at_MSC1, FTS1_at_MSC2,
FTS2_at_IEA1, FTS2_at_IEA2,
FTS2_at_MSC1, FTS2_at_MSC2,
FTS1_attached_to_MSC1, FTS1_attached_to_MSC2,
FTS1_attached_to_SSRMS, FTS1_attached_to_RSM,

FTS2_attached_to_MSC1, FTS2_attached_to_MSC2,
FTS2_attached_to_SSRMS, FTS2_attached_to_RSM,
ORU1_attached_to_IEA1, ORU1_attached_to_IEA2,
ORU1_attached_to_FTS1, ORU1_attached_to_FTS2,
ORU1_attached_to_MSC1, ORU1_attached_to_MSC2,
ORU2_attached_to_IEA1, ORU2_attached_to_IEA2,
ORU2_attached_to_FTS1, ORU2_attached_to_FTS2,
ORU2_attached_to_MSC1, ORU2_attached_to_MSC2,
ORU3_attached_to_IEA1, ORU3_attached_to_IEA2,
ORU3_attached_to_FTS1, ORU3_attached_to_FTS2,
ORU3_attached_to_MSC1, ORU3_attached_to_MSC2,
ORU4_attached_to_IEA1, ORU4_attached_to_IEA2,
ORU4_attached_to_FTS1, ORU4_attached_to_FTS2,
ORU4_attached_to_MSC1, ORU4_attached_to_MSC2}

The two IEAs (actually IEA pallets) serve as two general locations. One ORU can be attached to each IEA. An MSC can be near either one of the two IEAs. Each MSC holds one positioner: MSC1 holds the SSRMS, while MSC2 holds the RMS. Stowed on (or attached to) each MSC can be any number of ORUs and FTSs. Each FTS can be stowed on either MSC, or it can be attached to either positioner. The positioners act as positioning devices, placing the FTS it is holding at either MSC or either IEA. The FTSs are manipulating devices, which can remove or install an ORU. The MTs are transporting devices, which move MSCs near either IEA. MT1 transports only MSC1, and MT2 transports only MSC2; therefore, the MT/MSC pairs act as inseparable units—they are given separate symbols only for the sake of syntax.

To further describe the operation of the FTS environment, the interpretations of the verbs are given below. The use of these verbs differs slightly from their use in the Task Analysis Methodology document, but the spirit is the same.

transport: An MT transports an MSC to either of the two IEAs, if the positioner belonging to the MSC is not occupied (has not FTS attached to it).

position: A positioner can position an FTS at an IEA or an MSC, provided the MSC which holds the positioner is near the desired location. So, for instance, SSRMS can position FTS2 at IEA1 if MSC1 is near IEA1 and FTS2 is attached to SSRMS.

remove: A positioner can remove an FTS, if the FTS is stowed on an MSC, and the MSC which holds the positioner is near the MSC which holds the FTS. The result is that the FTS is attached to the positioner. Also, an FTS can remove an ORU, provided it is placed at the ORU's location and the ORU is not attached to another FTS. The result is that the ORU is attached to the FTS.

replace: A positioner can replace an FTS onto the MSC where the FTS is positioned. The FTS must, of course, be attached to the positioner, and it must be at one of the two MSCs. The result is that the FTS is attached to (or stowed on) the MSC where it is located. In addition, an FTS can replace an ORU onto either IEA or either MSC. The FTS must be holding the ORU, and if the FTS is at an IEA, that IEA must not have an ORU already attached to it. The result is that the ORU is attached to the IEA or MSC where the FTS is located.

The action sentence consists of an actor, a verb, a direct object and an indirect object; depending on the verb, the indirect object may be *null*. Following is a context-free grammar which generates all syntactically correct specific action sentences. The nonterminals are in italics, and the start symbol is *S*. If the indirect object is not specified,

it is *null*. Note that all of the nonterminals (except *S*) correspond to class names, which expand to one of their members.

S → MT1 transport MSC1 *IEA* | MT2 transport MSC2 *IEA* |
P position *FTS MSC* | *P* remove *FTS* | *FTS* remove *ORU* |
P replace *FTS* | *FTS* replace *ORU*

IEA → IEA1 | IEA2

FTS → FTS1 | FTS2

P → SSRMS | RSM

MSC → MSC1 | MSC2

ORU → ORU1 | ORU2 | ORU3 | ORU4

This grammar is used in exploration to generate random actions. Not all of these actions are applicable for a given environmental state, but part of the function of the rule base is to determine when an action is effective, i.e., when a particular action can be executed and produces some change in state.

It is rather obvious by their names which states in **Z** are pertinent to which actors and objects, i.e., which states are included in the mask for each actor and object. However, there is an exception: the states MSC1_near_IEA1 and MSC1_near_IEA2 are pertinent to both MSC1 and SSRMS, because when the SSRMS is positioning an FTS, it is necessary to know what is nearby. The states SSRMS_near_IEA1 and SSRMS_near_IEA2 could be added, but since the SSRMS is always located on MSC1, such states would be redundant. The same holds for the states MSC2_near_IEA1 and MSC2_near_IEA2.

Another exception is that the states FTS1_attached_to_SSRMS and FTS2_attached_to_SSRMS are pertinent not only to SSRMS, and to FTS1 and FTS2 (respectively), but also to MSC1. Although, strictly speaking, these states are not

states of MSC1, they are included because, if they were not, the rules for transporting the MSCs would all have low probability. Recall that, for an MT to transport an MSC, the positioner belonging to the MSC must not be carrying an FTS. Therefore, if these states are not included in the mask for the MSC, the rules for transporting could not distinguish between situations where the transport is effective or ineffective, so the probability of success would be fairly low and the rule would never be used in planning. The states FTS1_attached_to_RSM and FTS2_attached_to_RSM are considered pertinent to MSC2 for the same reason.

Some probabilistic elements are added to the environment, to test the ability of PRIME to trade off actions with differing probabilities of success. In the case of a positioner positioning an FTS, there are two effects which may occur: either the positioner succeeds in positioning the FTS at the desired location, or it fails and the environmental state does not change. Table 4.3 gives the probabilities of success of a positioning action, depending on the actor (the positioner) and the indirect object (the location). In addition to these probabilities, there is also a probability of 0.1 that FTS1 does not remove or replace an ORU at IEA2.

Table 4.3 Probability of success of positioning

	IEA1	IEA2	MSC1	MSC2
SSRMS	0.9	0.8	1.0	0.9
RSM	0.8	0.9	0.9	1.0

5. Experimental Results

PRIME was tested using the simulated environment described in §4.5. In this section, the results of some planning experiments are presented, as well as some data on execution time and memory consumption.

The rule base was initially developed with 10,000 random exploration cycles, where each cycle consisted of randomly generating a valid environmental state, choosing an action at random, executing the action, updating the rule base using the new environmental state, and generalizing the active and correct rules from this cycle. Following these 10,000 iterations, 2000 extra cycles were executed, identical to the previous cycles except that the action was taken from one of the active rules for the given environmental state. The purpose of these cycles was to refine the rules in the rule base, allowing their probabilities to converge somewhat, rather than to generate new rules.

Table 5.1 Training statistics

# iter	min:sec	sec/iter	# rules	# bytes	bytes/rule	# nodes	nodes/rule
1000	0:45	0.045	731	87972	120.3	2054	2.8
2000	1:44	0.052	1403	180236	128.5	3510	2.5
3000	5:51	0.117	2188	304212	139.0	5517	2.5
10000	115:39	0.695	7262	1223456	168.5	18114	2.5
12000	190:00	0.950	8879	1577400	177.7	21192	2.4

Table 5.1 gives the execution times (in CPU minutes and seconds on a SPARCstation 1) and rule base storage requirements at various stages in the training process. The execution time per iteration increases steadily with the number of iterations, because there are more rules to generalize in each cycle. The number of bytes per rule required to store

the rule base grows slowly with the number of rules, because the increasing number of general rules (which require multiple entries in the rule trie—see §4.1) require more storage. The number of trie nodes used to store the rules is better than linear in the number of rules.

The basic theme of all the planning tests in this section is exchanging a failed ORU attached to one of the IEAs with a replacement ORU found on one of the MSCs. This is a typical task in maintaining the Space Station Freedom, as illustrated in [25]. A standard starting state was chosen, which can be described as follows:

MSC1 near IEA1

MSC2 near IEA1

FTS1 at MSC1

FTS2 at MSC2

FTS1 attached to MSC1

FTS2 attached to MSC2

ORU1 attached to IEA1

ORU2 attached to IEA2

ORU3 attached to MSC1

ORU4 attached to MSC2

The ultimate goal is to replace ORU1 with ORU3, but, to avoid being too ambitious, a subgoal was first given to the planner:

ORU1 attached to MSC1

This serves to get the failed ORU out of the way, before the replacement ORU is attached. The plan generated was:

SSRMS remove FTS1

SSRMS position FTS1 IEA1

FTS1 remove ORU1

SSRMS position FTS1 MSC1

FTS1 replace ORU1

The plan was executed successfully, although a trace indicated that the second action was retried once, because the action was ineffective the first time. This was true of many of the actions whose effects were stochastic (as defined in Table 4.3); the retry loop in action execution proved to be very useful.

As encouraging as this was, the planner failed to achieve the next goal:

ORU3 attached to IEA1

Not even a partial plan was generated. To investigate where the deficiency lay, the goal was broken into several, more easily achieved subgoals, and each was tried in succession. The first goal was

ORU3 attached to FTS1

and the planner responded with

FTS1 remove ORU3

Then the goal was changed to

FTS1 at IEA1

and the planner responded instantly with

SSRMS position FTS1 IEA1

Finally, the planner was given the ultimate goal

ORU3 attached to FTS1

and it generated the partial plan

SSRMS position FTS1 MSC1

RSM remove FTS2

which, of course, failed. The conclusion is that there was no rule for attaching ORU3 onto IEA1 with FTS1 in that particular situation, and the planner could not find a nearby state which did provide a rule.

In order to try to learn a rule for this replacement, exploration was used. The machine explored its environment in closed-loop fashion, repeatedly executing actions and observing their effects, with no randomization of the state as was the case in training. A stopping condition was set so that exploration would cease when ORU3 was attached to IEA1. The machine executed 466 actions before stopping, and the environmental state was described by:

MSC1 near IEA1
MSC2 near IEA1
FTS1 at MSC1
FTS2 at IEA1
FTS1 attached to MSC1
FTS2 attached to SSRMS
ORU1 attached to MSC2
ORU2 attached to IEA2
ORU3 attached to IEA1
ORU4 attached to MSC2

The state was then reset to the state just before the failed plan, and planning was tried again, just to see if the exploration had any effect. It did not—the same faulty partial plan was generated. This may be because the state in which the replacement of ORU3 occurred during exploration was too far from the state in which the planning was attempted, so it could not use the new rule that was learned. This seems to indicate the need for goal-

directed exploration, in which reasonable actions are chosen based on the goal to be achieved, to keep the machine from wandering too far from the initial state.

Just to provide some indication of planning time, the state was reset to the standard starting state and the following goals were set:

FTS1 at IEA1

ORU1 not attached to IEA1

ORU3 attached to FTS1

The correct 7-step plan

SSRMS remove FTS1

SSRMS position FTS1 IEA1

FTS1 remove ORU1

SSRMS position FTS1 MSC1

FTS1 replace ORU1

FTS1 remove ORU3

SSRMS position FTS1 IEA1

was generated and executed in 1:16 (minutes:seconds) CPU time. The state was then reset to the standard starting state, and the goals were changed to:

FTS2 at IEA1

ORU1 not attached to IEA1

ORU3 attached to FTS2

attempting to force the planner to use FTS2 instead of FTS1. The following plan was generated and executed in 1:28 CPU time:

SSRMS remove FTS1

FTS1 remove ORU3

SSRMS position FTS1 MSC2

FTS1 replace ORU3

RSM remove FTS2

FTS2 remove ORU3

RSM position FTS2 IEA1

SSRMS position FTS1 IEA1

FTS1 remove ORU1

This plan may seem nonoptimal, but examination of the rule base showed that not all the rules necessary to generate the optimal plan were present. This incompleteness forces the planner to generate circuitous plans, in order to use the knowledge that does exist in the rule base. The above plan is indeed the optimal plan, based on the current knowledge of the machine.

From the resulting state after the above plan was executed, the goal

ORU3 attached to IEA1

was set, and the planner responded with:

SSRMS position FTS1 MSC1

FTS1 replace ORU1

SSRMS replace FTS1

MT1 transport MSC1 IEA2

SSRMS remove FTS1

FTS1 remove ORU1

It is apparent, from this example and the example in the beginning of this section, that there are no rules for replacing ORU3 in IEA1 with either FTS; this conclusion was verified by examining the rule base.

The state was again reset to the standard starting state, and a different goal was tried:

FTS1 at IEA2

ORU3 attached to IEA2

The plan generated was:

SSRMS remove FTS1

FTS1 remove ORU3

SSRMS position FTS1 MSC2

FTS1 replace ORU3

SSRMS replace FTS1

MT2 transport MSC2 IEA2

RSM remove FTS1

FTS1 remove ORU3

RSM position FTS1 IEA2

FTS1 replace ORU3

Note that incompleteness in the rule base forced the planner to generate a longer plan than strictly necessary. This plan failed because ORU1 was not removed before trying to insert ORU3 into IEA2. This was caused by the use of an incorrect general rule where there were no specific rules. Since PRIME stored the failure of the last action as a specific exception to the general rule, planning was attempted again, with the following result:

MT1 transport MSC1 IEA1

RSM position FTS1 MSC2

FTS1 replace ORU3

The planner was forced by this new exception to try a different rule, which was active under different conditions, so the machine had to move to a different state. Finally, the goal was changed to

FTS1 at IEA2

ORU2 not attached to IEA2

ORU3 attached to IEA2

making the requirement of the removal of ORU2 explicit. The planner responded with

RSM position FTS1 IEA2

MT1 transport MSC1 IEA1

which was worthless. Apparently, there was no rule for removing ORU2 from IEA2 with FTS1.

Many other planning problems were tried, changing the goals and changing the starting state, but most led to the same results: planning failed because the rule base was incomplete. In addition, all "nonoptimal" plans have been due to incompleteness in the rule base, and not any ineffectiveness in the planning. In particular, no example was found where being more specific in the goal has resulted in discovering a better plan. Also, replanning after executing an unsuccessful partial plan has never resulted in completing a successful plan, and no guiding of the planner with subgoals has resulted in a sequence which would not be generated anyway. These statements imply that the likelihood of finding a successful plan does not vary with the length of the plan.

All successful plans were found in 5 seconds to 2 minutes; a search which lasted longer than that always resulted in failure. This may indicate that the heuristic is a useful guide for the search, but the link weight may not be strong enough. However, the link weight cannot be strengthened (made lower in value) without jeopardizing the planner's ability to generate long plans. The problem may be with the width of the planning tree, not the depth; however, the number of children nodes averaged about 4, and never exceeded 10—these quantities do not seem unreasonable.

In an attempt to enrich the rule base, more exploration was used to further train the machine. An extra 3,000 iterations were executed in about 1.5 hours, and the resulting rule base contained 11,720 rules and occupied 2,107,884 bytes. Nevertheless, this larger rule

base still demonstrated numerous gaps of knowledge which caused planning to fail. It seems that, if the rule base for this environment is to be at all robust, it would need about four times the number of rules it has now; this would consume too much memory to be useful. If PRIME is to work in an environment of this size or larger, the storage requirements of the rule base need to be greatly reduced.

In order to continue experimentation, the environment was reduced by making the following changes:

1. The MSCs are always near IEA1; there is no transporting of MSCs. ORU4 is permanently attached to IEA2.
2. Any environmental state formed by the random state generator has FTS1 attached to either MSC1 or SSRMS. Similarly, any randomly generated state has FTS2 attached to either MSC2 or RSM.
3. For any randomly generated positioning action, only SSRMS can position FTS1, and only RSM can position FTS2.

To generate the new rule base, random exploration was run for 20,000 iterations, using random actions. This training period lasted 4 hours (this time was shorter because the code was optimized) and generated 10,241 rules which occupied 2,016,356 bytes. Because the number of actions and the number of possible environmental states were reduced, the machine had a good amount of exposure to all actions in this training period, as evidenced by the learning rate α stored with each rule—a low α indicates numerous exposures to the rule, because α decays every time the rule is updated. The rules for removing and replacing ORUs received less exposure than the rules for positioning, but this is acceptable, since it is in positioning that the environment is stochastic.

A new standard starting state was devised:

MSC1 near IEA1

MSC2 near IEA1

FTS1 at MSC1

FTS2 at MSC2

FTS1 attached to SSRMS

FTS2 attached to RSM

ORU1 attached to MSC1

ORU2 attached to MSC2

ORU3 attached to IEA1

ORU4 attached to IEA2

The environment was set to this new starting state, and the goal was set to

ORU3 attached to MSC1

The derived plan was:

FTS1 remove ORU1

SSRMS position FTS1 MSC2

FTS1 replace ORU1

SSRMS position FTS1 IEA1

FTS1 remove ORU3

SSRMS position FTS1 MSC1

FTS1 replace ORU3

This shows that there was still some incompleteness in the rule base. In fact, when the goal was then set to

ORU1 attached to IEA1

the response from the planner was

RSM replace FTS2

The rule base obviously needed some additional training, so the random action generator was modified to emphasize removing and replacing of ORUs, and 10,000 iterations of exploration were executed. The rules now numbered 11,821 and took 2,244,916 bytes of memory. The rules showed much better exposure to the removing and replacing actions: the values of α for rules for these actions averaged around 0.1, and many other rules had the minimum value of α, 0.015.

The new starting state was set again, and the goal was reset to

ORU3 attached to MSC1

The planner generated the same successful but circuitous plan it did before the additional 10,000 exploration cycles. When the goal was set to

ORU1 attached to IEA1

the planner responded with a different but equally ineffective plan:

RSM remove FTS2

FTS2 remove ORU2

SSRMS replace FTS1

The rule base did not seem any better than before the additional training.

The state was reset to the new starting state, and the goal was specified as

ORU3 attached to RSM

The plan to achieve this goal was:

SSRMS replace FTS1

RSM replace FTS2

RSM remove FTS1

SSRMS remove FTS2

RSM position FTS2 IEA1

FTS2 remove ORU3

This plan failed, of course, because the RSM no longer held FTS2 when the position action was executed. The plan was considered acceptable by the planner, because the environment entered into a previously unencountered state when the two actions

RSM remove FTS1

SSRMS remove FTS2

were executed, thereby forcing the planner to subsequently use general rules, which happened to be incorrect for that state.

However, the real problem was discovered only after examining the rule base. The planner did not take the obvious route, not because the rule for RSM position FTS2 IEA1 in the start state was missing, but because it had a probability of success of 0.743, which is below the planner's probability threshold of 0.75. This raises the following difficulty: once the probability of success of a rule falls below the threshold for use in planning, it will never be used again in the normal operation of PRIME (which consists primarily of planning), so its probability will never rise. This rule can only be used in exploration, and exploration provides the only chance to raise the probability estimate. Thus, periodic exploration is necessary in order to keep the rule base current. This seems entirely impractical for an autonomous machine operating in a real environment, unless goal-directed exploration is used, to keep the machine from deviating too far from the current state or the goal. In any case, execution of plans clearly provides insufficient experience for maintaining the rule base.

6. Future Work

This section discusses some topics which could be addressed in future research in extending the capabilities of PRIME.

6.1 Integrating Boltzmann machines

Moed [21, 22] has proposed two Boltzmann machine architectures for use in the Organization Level of Saridis' Hierarchical Intelligent Control System. A Boltzmann machine is a type of neural network model which uses energy minimization to determine the output values of its nodes. In [23] Moed proposes a modified genetic algorithm for searching for the minimum energy configuration of a Boltzmann machine; this algorithm is proven to converge in probability to the global minimum. A primary thrust of future work should be the integration of these Boltzmann machines into the operation of PRIME. One of Moed's proposed architectures, Associative Rule Memory [22], attempts to learn the effects of specific actions and the probability of these effects. The other architecture, called the Complexity Model [21], attempts to learn the complexity of executing specific actions under certain environmental states, as indicated by the entropy feedback from the coordination level.

The Associative Rule Memory has six levels of nodes, which represent:

1. object name in desired effect
2. state of object in desired effect
3. actor in action sentence
4. verb in action sentence
5. direct object in action sentence
6. indirect object in action sentence.

Each node in the object name level corresponds to an object in the environment. Each node in the state level corresponds to an object state. The number of state level nodes may be the same as the number of object states (N), or, if the states for each object are compiled using the mask string, the state level need only contain the number of nodes equal to the maximum number of relevant states for an object, taken over all objects. In each of the remaining levels, there is one node per member of the sets **A**, **V**, **U**, and **U**, respectively.

To use this network for associative recall, exactly one node in each of the first two levels is asserted, corresponding to the object and the state of that object which needs to be changed. These levels are then fixed, and the other levels are allowed to vary (subject to constraints) during the settling of the network. Once the network has found a configuration with minimum energy, the output of the network is the action sentence indicated by the nodes asserted in the last four levels, and the energy of the network is related to the probability of that action changing the state of the object denoted by the first two levels. Therefore, the network selects the action with the highest probability of achieving a desired effect.

This network cannot provide the entire function of the Organization Level, for two reasons. First, it must be trained with a set of symbolic rules, such as those which comprise PRIME's rule base. Second, as it stands, it is insufficient for representing the entire rule base, because only one effect at a time is considered for any given action, and there is no accounting for the conditions under which these actions may be performed. The effect of an action on the environment may depend on the environmental state in which it is executed, but this cannot be represented in the Associative Rule Memory.

Nevertheless, the Associative Rule Memory may have two important roles in PRIME: implementing goal-directed exploration, and increasing the efficiency of planning. During exploration, PRIME currently selects actions and rules to test entirely randomly. This is

ineffective if one is trying to develop the rules required to achieve a particular goal. However, if this network is used to suggest a set of actions to try, the machine may avoid executing actions which do not serve the desired purpose. Of course, in order to suggest a particular action which achieves a desired effect, the network must have been exposed to experience under other conditions in which that action did result in the desired effect; even so, goal-directed exploration is a great improvement over random exploration, and the Associative Rule Memory can certainly help achieve this function. Goal-directed exploration is discussed further in the next section.

The Associative Rule Memory may increase the efficiency of planning in PRIME by suggesting actions which should be tried when expanding a node. Currently, the A* planner uses *all* active rules with high probability and low generality when generating links to expand a node. If the Associative Rule Memory is used to eliminate (or at least give low preference to) rules whose actions do not appear to lead toward the goal, this could prevent the planner from following wayward paths due to an inaccurate heuristic function.

The Complexity Model is similar to the Associative Rule Memory, in that it has a state level and four action levels corresponding to the four components of an action sentence, but it has some hidden levels as well. Its primary purpose is to predict the complexity of an action under a particular environmental state, according to the history of entropy feedback from the Coordination Level. It can also be used to find a set of actions which have low complexity for a given state—it can even be settled in conjunction with the Associative Rule Memory to find low-complexity actions which have a high probability of affecting a particular object state under the given conditions. If complexity values are used at all in the Organization Level, the role of the complexity model in PRIME would primarily be to predict the complexity of an action which appears in a general rule The complexity stored

in the specific rules would always be preferred over the Complexity Model prediction, but the Complexity Model may be useful in situations where only a general rule can be applied.

The rule representation in PRIME is ideal for training both of these Boltzmann machines, since the environmental state and the conditions and effects of the rules are binary strings (with some don't cares included). This is one of the reasons why the classifier-type representation was chosen.

6.2 Goal-directed exploration

Goal-directed exploration is a modification of the random exploration currently implemented in PRIME. In random exploration, an action is generated randomly or an active rule is selected randomly, the action is executed, and the results are used to modify the rule base. In goal-directed exploration, a goal is specified, and the next action to execute is selected based on the likelihood that the action will lead to the satisfaction of the goal.

Goal-directed exploration has several advantages over random exploration. It simplifies the development of the rule base by allowing a supervisor more control over the training process. If the supervisor discovers that the rule base is deficient in some area, he or she may specify a goal which the machine must satisfy, and allow goal-directed exploration to guide the choice of actions to try. In random exploration, most of the actions selected either do not lead to a desired state or have no effect at all; this results in a proliferation of rules specifying no change in state, which is a waste of memory in most cases. The use of goal-directed exploration may help the machine to avoid these ineffective actions, thereby saving memory as well as providing direction in training.

Goal-directed exploration also provides a useful automatic recovery mechanism when planning fails. If the derived plan does not satisfy the desired goal, the machine may attempt to satisfy the goal by exploration. Even if the goal still is not satisfied after a period of time, this exploration might provide enough information to allow planning to continue.

Finally, §5 pointed out that the machine needs to explore its environment periodically in order to keep its rule base current. It is not clear, however, when exploration should be employed during operation in the field, aside from a recovery situation. Perhaps, if the goal is not imperative, the machine could use goal-directed exploration for a while, before engaging in planning. In any case, the exploration would probably need to be goal-directed, in order to keep the machine from wandering too far from the goal.

6.3 Rule representation

There are several issues related to the representation of the rule base in PRIME which call for further research. Foremost is the representation and generation of general rules. Once a general rule is created, it is retained in the rule base unless its support drops below a threshold, in which case it is deleted. Since general rules are created from combinations of more specific rules, a highly general rule will have many more specific general rules which are simply partial instantiations of it. In most cases, this is clearly a waste of space, and the same functionality could be achieved by removing these more specific general rules. However, they are retained for two reasons. First, the more specific rules will usually have more accurate estimates of the probability of effect; in fact, there can be many general rules which predict the same effect with different probabilities, but there is only one most specific rule (even if it is a general rule), and the probability from this rule is used as the estimate of the probability of effect. Second, if circumstances lead to the deletion of a

highly general rule, due to loss of support, it is useful to retain the more specific versions of the general rule which have not lost support. It appears, therefore, that another method of generating and using general rules is needed, if this wasted space is to be recovered.

Another weakness in the present generalization method is that there is no way to prevent the repeated regeneration of general rules which already exist in the rule base. In addition, there is the problem of class interaction, which limits the power of action generalization. This problem hinders the usefulness of general-action rules for filling in gaps of knowledge which are not covered by specific experience. In fact, experimentation shows that the general rules are not as effective in this role as expected. Some investigation of the effects of varying the create and delete thresholds in the generalization process is needed, but it appears initially that it may be advantageous to represent the general rules in some sort of associative memory (neural network or some other architecture), which responds with the closest rule for the current state and desired action. This might require less memory than explicitly storing general rules, and it might provide a greater extension of the machine's knowledge than the present generalization scheme does.

Because action generalization is a rather ineffective method of generalizing rules, and because the memory saved by reducing the size of the condition string is negligible compared to the extra memory used by redundant general rules, the assumption of locality of effect is no longer useful. This is especially true in light of the fact that locality of effect reintroduces the frame problem. Therefore, future methods of rule representation should attempt to avoid this limiting assumption.

6.4 Probability estimation

One area of future work in the estimation of effect probabilities is the derivation of expressions for the mean square error of the estimate with various families of time-varying learning rates $\alpha(t)$. These expressions would be more useful than the steady-state (constant α) results in predicting the effects of various parameters on the estimation procedure. Another path of research might be the use of an adaptation technique to vary some parameter in the estimation scheme, such as the decay rate of α.

6.5 Skill development

A skill can be defined as a sequence of actions whose effects are well known and highly certain. In PRIME, a skill might take the form of a string of rules which reliably predict the effects of their actions when used in sequence. Skills provide an isolation of rules to a certain context, allowing the probability estimates to more accurately reflect the physical probabilities, if they are in fact context-dependent. This effectively relaxes the assumption that the environment is a Markov process, allowing PRIME to model an environment with memory, or to increase the accuracy of its model if the machine does not have access to all the necessary states in the environment. In this way, the use of skills may supplement the role of the effect probabilities in modelling an uncertain environment. In addition, skills can simplify planning by acting as larger building blocks which can achieve distant subgoals in one "step".

There are many questions about skill development which should be answered before this idea can be implemented. The primary one is: How should one choose the rules to be grouped as a skill? One solution is to maintain temporal links between rules, with strengths associated with the links. These strengths could represent the frequency of execution one

rule after another; in this case, the strength would increase if the second rule is executed after the first, and it would decrease some other rule was executed after the first. Alternatively, the link strength could represent the likelihood that the second rule will correctly predict its effect when executed after the first rule; thus, the strength would increase whenever the second rule was successful following the first rule, and it would decrease whenever the second rule failed following the first rule. This scheme assumes that the first rule, for the most part, provides the context for the success of the second rule; if this is not so, it is uncertain whether this method will produce any useful results.

If the temporal links are used, strong chains of rules may be recognized and isolated to form a skill. This raises the question: Should rules in skills actually be isolated, in the sense that probability updates of rules in skills are separate from updates of the same rules outside of skills? If these probabilities are truly context-dependent, then isolation may permit more accurate probability estimation. However, if a rule should suddenly and repeatedly fail outside of a skill, it might be wise to doubt the certainty of the same rule within the skill the next time the skill is executed.

Once a skill is formed, it may be treated as another rule by generating condition and effect strings for it. This could be accomplished by tracing through the sequence, keeping track of the required condition values and the ultimate effects of the sequence. The probability of the skill could be the product of the probabilities of the individual rules in the skill. Then the skill could be used in planning just like any other rule. In fact, the temporal links themselves might be used in planning, giving preference to rules with higher link strengths.

When a skill is executed, presumably as a member of a plan, it could be passed a subgoal. As each rule in the sequence is about to be executed, its contribution to the satisfaction of the goal could be evaluated; if it makes no contribution, it may be skipped.

(The subgoal would probably come from the planner itself—it would not be difficult to derive such a subgoal from a backward-chaining planner, for instance.) This procedure enables the machine to avoid executing superfluous actions.

There are many other questions about skill development which must be answered. How and when is a skill deleted? How is a skill honed, i.e., how are superfluous rules removed and new rules added? How and when should skills be merged or divided? Should conditional paths be allowed in skills? If so, how are they developed? These questions, and the discussion above, can provide a basis for investigating the implementation of skill development as a powerful extension of PRIME.

7. Discussion and Conclusions

Unfortunately, the results of experimentation presented in §5 were mostly negative; even the positive results were tempered by associated negative ones. For instance, the execution times for rule recovery, exploration and successful planning were very reasonable, although a better method must be found for aborting an unsuccessful planning process—the planner has been known to take as long as 10-15 minutes to conclude that there is no plan to achieve the goal. The storage requirement per rule was also respectable. Nonetheless, the total memory consumed by the rule base was much too large for the amount of coverage of the domain it provided—it proved difficult to achieve a workable rule base within 2 megabytes.

Much of this space was wasted by redundant general rules, subsumed by other general rules, and much of the time in generalization was wasted on rederiving general rules which were already present in the rule base. Furthermore, there were many cases where the application of a general rule (especially a general action rule) was inappropriate, leading to an incorrect prediction of the effect of an action. Although PRIME learns the exceptions to these rules through experience, and a mistake is only made once in any situation, it seems clear that a different approach to rule generalization is required, one which provides better coverage of the domain and more accuracy, without consuming memory and time with redundancies. Perhaps a neural network or other type of associative memory is in order.

Another inefficiency in the rule base is the numerous rules which dictate when an action has no effect. A certain number of these are necessary and inevitable, but an improved method of rule development, such as goal-directed exploration, might learn to avoid ineffective actions. As it stands, PRIME is not very effective at generating a rule base from scratch, although it might be useful for updating a preexisting rule base. Goal-

directed exploration, however, may provide the degree of control necessary to effectively generate a useful rule base. These goals can be provided by a human supervisor or by a program, if the appropriate training goals are known in advance. Goal-directed exploration would also be useful in recovery situations, when planning fails, in order to gather new information, and it would certainly be more well-behaved than random exploration when used periodically to keep the rule base current.

It is interesting to note that the assumption of a stochastic environment has forced PRIME to become stochastic itself, in two ways. First, it is established in §2.2 that the probability estimates must have a nonzero variance in order to respond to a nonstationary environment, thereby maintaining a small level of noise in the system. Second, §5 and §6.2 discussed the need for periodic exploration of the environment, to keep some of the rules from stagnating if their probabilities dip below the planner's threshold. This manifests itself as occasional randomness or unpredictability in the behavior of the machine.

All facets of PRIME could stand some improvement, in addition to those listed above. Additional work is required in order to understand and specify the transient behavior of the probability estimation procedure, perhaps to achieve quicker convergence to the physical probabilities. The current planner may be adequate for the environment studied here, but more sophisticated environments will require more sophisticated planning techniques, such as hierarchical planning or planning with parallel paths. Temporal planning would be especially useful if dynamic environments are considered. However, the basic methodology of PRIME provides the necessary tools for allowing a machine to operate in a stochastic or uncertain environment with a minimum of assumed *a priori* knowledge, while maintaining the validity of its knowledge in order to effectively achieve required goals.

References

[1] Anderson, J. A. *The Architecture of Cognition.* Harvard University Press, Cambridge, MA, 1983.

[2] Anderson, J. A. and Rosenfeld, E. *Neurocomputing: Foundations of Research.* MIT Press, Cambridge, MA, 1988.

[3] Barto, A. G. and Anandan, P. Pattern-recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15(3):360-375, 1985.

[4] Carbonell, J. G. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In Michalski, R. S., Carbonell, J. G., and Mitchell, T.M. (eds.), *Machine Learning: An Artifical Intelligence Approach, Volume II*, Morgan Kaufmann, Los Altos, CA, 1983.

[5] Chow, Y. S. and Yu, K. F. The performance of a sequential procedure for the estimation of the mean. *The Annals of Statistics*, 9(1):184-189, 1981.

[6] DeJong, G. An approach to learning from obsercation. In Michalski, R. S., Carbonell, J. G., and Mitchell, T.M. (eds.), *Machine Learning: An Artifical Intelligence Approach, Volume II*, Morgan Kaufmann, Los Altos, CA, 1983.

[7] Dolan, C. P. and Dyer, M. G. Symbolic schemata, role binding and the evolution of structure in connectionist memories. In *IEEE First International Conference on Neural Networks*, San Diego, CA, 1987.

[8] Ernst, G. and Newell, A. *GPS: A Case Study in Generality and Problem Solving.* ACM Monograph Series. Academic Press, New York, NY, 1969.

[9] Fikes, R. E., Hart, P. E., and Nilsson, N. J. Learning and executing generalized robot plans. *Artificial Intelligence*, 3, 1972.

[10] Fikes, R. E. and Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189-208, 1971.

[11] Ghosh, M. and Mukhopadhyay, N. Sequential point estimation of the mean when the distribution is unspecified. *Communications in Statistics—Theory and Methods*, A8(7):637-652, 1979.

[12] Good, I. J. *The Estimation of Probabilities: An Essay on Modern Bayesian Methods.* MIT Press, Cambridge, MA, 1965.

[13] Green, C. Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence*, Washington, DC, 1969.

[14] Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100-107, 1968.

[15] Holland, J. H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

[16] Holland, J. H. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, R. S., Carbonell, J. G., and Mitchell, T.M. (eds.), *Machine Learning: An Artifical Intelligence Approach, Volume II*, Morgan Kaufmann, Los Altos, CA, 1983.

[17] Holland, J. H., Holyoak, K. J., Nisbett, R. E., and Thagard, P. R. *Induction: Processes of Inference, Learning, and Discovery*. MIT Press, Cambridge, MA, 1986.

[18] Kirkpatrick, S., Gelatt, C. D. Jr., and Vecchi, M. P. Optimization by simulated annealing. *Science*, 220:671-680, 1983.

[19] Lippmann, R. P. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pp. 4-22, April 1987.

[20] Michalski, R. S. A theory and methodology of inductive learning. In Michalski, R. S., Carbonell, J. G., and Mitchell, T.M. (eds.), *Machine Learning: An Artifical Intelligence Approach, Volume I*, Morgan Kaufmann, Los Altos, CA, 1983.

[21] Moed, M. C. *The Organizer: Planning Tasks with an Emergent Connectionist/Symbolic System.* CIRSSE Document #42, Department of Electrical, Computer and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY, 1989.

[22] Moed, M. C. *Planning Tasks with an Emergent Connectionist/Symbolic System.* CIRSSE Document #46, Department of Electrical, Computer and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY, 1989.

[23] Moed, M. C. A Boltzmann machine for the organization of intelligent machines. To appear in *IEEE Transactions on Systems, Man, and Cybernetics*, 20(5), 1990.

[24] Narendra, K. S. and Thathachar, M. A. L. Learning automata—a survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4(4):323-334, 1974.

[25] NASA Flight Telerobotic Server Mission Utilization Team. *Flight Telerobotic Servicer Task Analysis Methodology*. Goddard Space Flight Center, April 1989.

[26] Nilsson, N. J. *Principles of Artificial Intelligence*. Morgan Kaufmann, Los Altos, CA, 1980.

[27] Robbins, H. and Monro, S. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400, 1951.

[28] Rumelhart, D. E. and McClelland, J. L. (eds.). *Parallel Distributed Processing, Volume 1*. MIT Press, Cambridge, MA, 1986.

[29] Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. In *Third International Joint Conference on Artificial Intelligence*, Stanford, CA, 1973.

[30] Sacerdoti, E. D. The nonlinear nature of plans. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, 1975.

[31] Saridis, G. N. *Self-Organizing Control of Stochastic Systems*. Marcel Dekker, New York, NY, 1977.

[32] Saridis, G. N. Toward the realization of intelligent controls. *IEEE Proceedings*, 67(8), 1979.

[33] Saridis, G. N. On the revised theory of intelligent machines. In *Proceedings of an International Workshop on Intelligent Robots and Systems*, Tsukuba, Japan, 1989.

[34] Sedgewick, R. *Algorithms*. Addison-Wesley, Reading, MA, 1983.

[35] Shastri, L. and Ajjanagadde, V. *A Connectionist System for Rule Based Reasoning with Multi-place Predicates and Variables*. Technical Report MS-CIS-89-06, University of Pennsylvania, Philadelphia, PA, 1989.

[36] Shoham, Y. What is the frame problem? In Georgeff, M. P. and Lansky, A. L. (eds.), *Reasoning About Actions and Plans*, Morgan Kaufmann, Los Altos, CA, 1986.

[37] Siklóssy, L. and Roach, J. Model verification and improvement using DISPROVER. *Artificial Intelligence*, 6:41-52, 1975.

[38] Starr, N. On the asymptotic efficiency of a sequential procedure for estimating the mean. *Annals of Mathematical Statistics*, 37:1173-1185, 1966.

[39] Stefik, M. Planning with constraints (MOLGEN: parts 1 and 2). *Artificial Intelligence*, 16, 1981.

[40] Touretzky, D. S. Representing conceptual structures in a neural network. In *IEEE First International Conference on Neural Networks*, San Diego, CA, 1987.

[41] Touretzky, D. S. and Hinton, G. E. Symbols among the neurons: Details of a connectionist inference architecture. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Los Angeles, CA, 1985.

[42] Williams, R. J. *Toward a Theory of Reinforcement-Learning Connectionist Systems*. Technical Report NU-CCS-88-3, College of Computer Science, Northeastern University, Boston, MA, 1988.

[43] Wilson, S. W. and Goldberg, D. E. A critical review of classifier systems. In *Proceedings of an International Conference on Genetic Algorithms*, Fairfax, VA, 1989.

[44] Zhou, H. H. *CSM: A Genetic Classifier System for Learning by Analogy*. PhD thesis, University of Michigan, Ann Arbor, MI, 1987.

Appendix A

Mean and Variance of Linear Reinforcement

Let $\{x[n]\}$ be a sequence of Bernoulli random variables, where $x[n] = 1$ if the effect in question occurred at time n. (The subscript i denoting the effect in question will be omitted to ease notation.) Let q be $P\{x[n] = 1\}\ \forall n$, i.e., the physical probability of the effect occurring. The probability q is constant, so these results are for stationary environments only. Let $\{p[n]\}$ be the sequence of effect probability estimates, whose dynamics are defined by (2.3a-b). These two equations can be condensed to one:

$$p[n+1] = (1-\alpha)p[n] + \alpha x[n]$$

This, in turn, can be expressed in closed form:

$$p[n] = (1-\alpha)^n p[0] + \sum_{k=1}^{n}(1-\alpha)^{n-k}\alpha x[k] \tag{A.1}$$

where $p[0]$ is the initial value of the estimate.

Since $E\{x[n]\} = q\ \forall n$,

$$E\{p[n]\} = (1-\alpha)^n p[0] + \alpha q \sum_{k=1}^{n}(1-\alpha)^{n-k} \tag{A.2}$$

The sum in the right term of (A.2) can be simplified:

$$\alpha\sum_{k=1}^{n}(1-\alpha)^{n-k} = \alpha\sum_{k=0}^{n-1}(1-\alpha)^{k} = 1 - (1-\alpha)^n \tag{A.3}$$

This can be proven by a change of variable. Let $\beta = 1-\alpha$; then

$$\alpha\sum_{k=0}^{n-1}(1-\alpha)^{k} + (1-\alpha)^n - 1$$

$$= (1-\beta)\sum_{k=0}^{n-1}\beta^{k} + \beta^n - 1$$

$$= 1 - \beta^n + \beta^n - 1 = 0$$

Substituting (A.3) into (A.2) we obtain

$$E\{p[n]\} = \mu_p[n] = q + (1-\alpha)^n(p[0] - q) \tag{A.4}$$

and

$$\lim_{n\to\infty} E\{p[n]\} = q \tag{A.5}$$

The variance of the estimate is given by

$$\sigma_p^2[n] = E\{(p[n] - \mu_p[n])^2\} = E\{p^2[n]\} - \mu_p^2[n]$$

Squaring (A.1) yields:

$$p^2[n] = (1-\alpha)^{2n}p^2[0] + \alpha^2\left(\sum_{k=1}^{n}(1-\alpha)^{n-k}x[k]\right)^2$$

$$+ 2\alpha p[0](1-\alpha)^n\sum_{k=1}^{n}(1-\alpha)^{n-k}x[k]$$

$$= (1-\alpha)^{2n}p^2[0] + \alpha^2\left(\sum_{k=1}^{n}(1-\alpha)^{2(n-k)}x[k] + \sum_{i=1}^{n}\sum_{j\neq i}(1-\alpha)^{2n-i-j}x[i]x[j]\right)$$

$$+ 2\alpha p[0](1-\alpha)^n\sum_{k=1}^{n}(1-\alpha)^{n-k}x[k]$$

Replacing k with n-k, and taking the expectation:

$$E\{p^2[n]\} = (1-\alpha)^{2n}p^2[0] + \alpha^2 q\sum_{k=0}^{n-1}(1-\alpha)^{2k} + \alpha^2 q^2\sum_{i=0}^{n-1}\sum_{j\neq i}(1-\alpha)^{i+j}$$

$$+ 2\alpha p[0](1-\alpha)^n q\sum_{k=0}^{n-1}(1-\alpha)^k \tag{A.6}$$

Squaring (A.4) yields:

$$\mu_p^2[n] = q^2 + (1-\alpha)^{2n}(p[0]-q)^2 + 2q(1-\alpha)^n(p[0]-q)$$

$$= q^2 + (1-\alpha)^{2n}(p[0]-q)^2 + 2p[0]q(1-\alpha)^n - 2q^2(1-\alpha)^n \tag{A.7}$$

Now we evaluate the difference between (A.6) and (A.7) term by term. Taking the last term of (A.6) minus the third term of (A.7) gives

$$2p[0]q(1-\alpha)^n\left(\alpha\sum_{k=0}^{n-1}(1-\alpha)^k - 1\right)$$

and substituting from (A.3), this reduces to

$$-2p[0]q(1-\alpha)^{2n}$$

Adding to this the first term of (A.6) and subtracting the second term of (A.7) results in

$$(1-\alpha)^{2n}\left(-2p[0]q + p^2[0] - (p[0]-q)^2\right)$$

$$= -q^2(1-\alpha)^{2n}$$

Subtracting the first and fourth terms of (A.7) yields

$$-q^2\left((1-\alpha)^{2n} - 2(1-\alpha)^n + 1\right)$$

$$= -q^2\left(1 - (1-\alpha)^n\right)^2$$

Again substituting from (A.3) and expanding, this becomes

$$-q^2\left(\alpha\sum_{k=0}^{n-1}(1-\alpha)^k\right)^2$$

$$= -q^2\alpha^2\sum_{k=0}^{n-1}(1-\alpha)^{2k} - q^2\alpha^2\sum_{i=0}^{n-1}\sum_{j\neq i}(1-\alpha)^{i+j}$$

Finally, adding in the second and third terms of (A.6) gives the expression for the variance:

$$\sigma_p^2[n] = \alpha^2 q(1-q)\sum_{k=0}^{n-1}(1-\alpha)^{2k} \qquad \text{(A.8)}$$

To find the limit of the variance, we use the following identity for the geometric sum:

$$\lim_{n\to\infty}\sum_{k=0}^{n-1}(1-\alpha)^{2k} = \frac{1}{1 - (1-\alpha)^2} = \frac{1}{\alpha(2-\alpha)}$$

Taking the limit of (A.8) and substituting the above expression yields

$$\lim_{n\to\infty} \sigma_p^2[n] = \frac{\alpha}{2-\alpha} q(1-q) = \frac{\alpha}{2-\alpha} \sigma_x^2 \tag{A.9}$$

because x is a Bernoulli random variable.

Appendix B

Storage of Full-Depth Radix Search Trie

This appendix evaluates the best- and worst-case storage requirements of a full-depth radix search trie for storing and recalling a set of objects with binary search keys. A full-depth trie is a tree of constant height, given by the length of the key, whose terminal nodes point to the appropriate objects to be recalled.

Let N be the number of objects to be stored. Let each of those objects have a binary key of constant length m bits. Let n be the size of the radix, i.e., the number of bits to be compared at once. Let p be the number of branches from any node in the trie; p is therefore 2^n.

A worst-case full-depth trie is one in which the keys of the objects are maximally sparsely distributed, thereby requiring the maximum possible number of nodes in the search trie. A worst-case trie for N=6, m=12, n=2, p=4 is illustrated in Figure B.1, where the squares are the objects being stored.

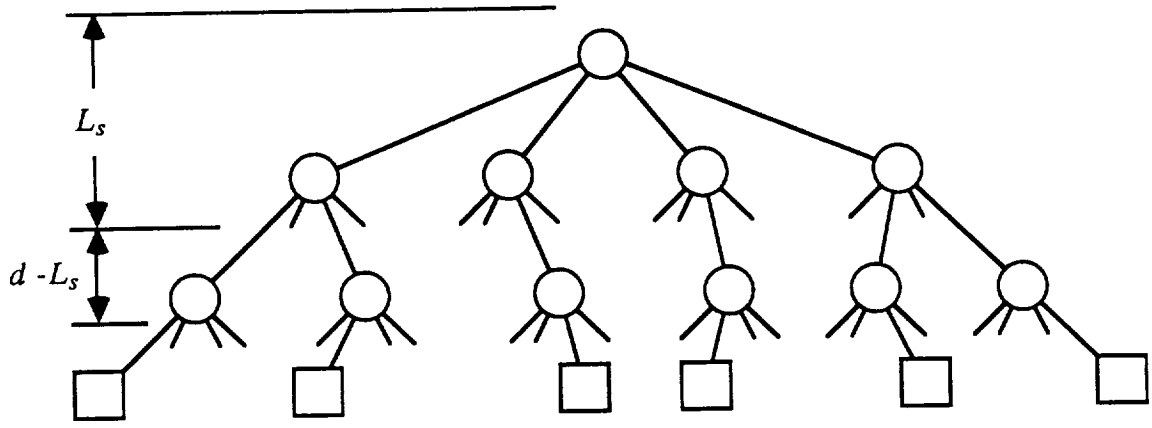

Figure B.1 Worst-case full-depth trie

The depth of any full-depth trie is given by

$$d = \text{rup}\left(\frac{m}{n}\right)$$

where the function rup(·) returns the smallest integer greater than its argument—it *rounds up* its argument. The number of shared levels in a worst-case trie is given by

$$L_s = \text{rup}(\log_p N)$$

A level is *shared* if it contains at least one node which may lead to the selection of more than one object. In Figure B.1, $L_s = 2$. The total number of shared nodes is then

$$N_s = \sum_{i=0}^{L_s-1} p^i = p^{L_s} - 1$$

The number of nodes in the unshared levels is simply

$$N_u = N(d-L_s)$$

so the total number of nodes in a worst-case full-depth trie is

$$N_{wc} = N_s + N_u = p^{L_s} - 1 + N(d-L_s)$$

Since $N_s = O(N)$, N_{wc} can be approximated as

$$N_{wc} = O\left(N + N\left(\frac{m}{n} - \log_p N\right)\right) \qquad \text{(B.1)}$$

For a best-case trie, the keys of the objects are as densely distributed as possible—they are clustered together, so they require the minimum number of nodes in the search trie. A best-case full-depth trie for N=7, m=5, n=1, p=2 is shown in Figure B.2. The tree consists of two parts, one which is completely filled, and the other which is a string of extra nodes necessary to complete the depth of the tree. The number of levels in the filled part of the tree is

$$L_f = L_s = \text{rup}(\log_p N)$$

so the number of nodes in the filled part is

$$N_f \le N_s = p^{L_f} - 1$$

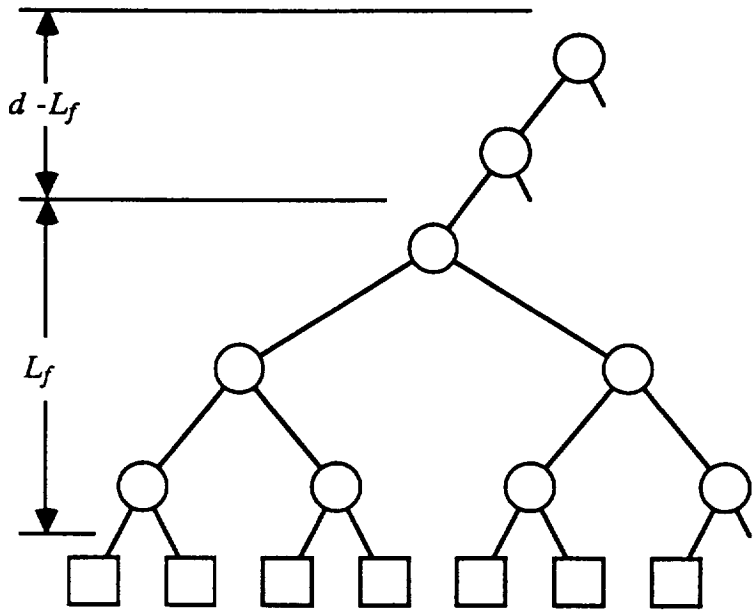

Figure B.2 Best-case full-depth trie

The inequality exists because a completely filled tree may not be necessary to store N objects. The number of extra nodes at the top of the tree is given by

$$N_e = d - L_f$$

The total number of nodes in a best-case full-depth trie is therefore

$$N_{bc} = N_f + N_e = p^{L_f} - 1 + d - L_f \tag{B.2}$$

which is approximately

$$N_{bc} = O(N)$$

for N so large that N_e is negligible.

Appendix C

Storage of Compact Binary Radix Search Trie

This appendix evaluates the best- and worst-case storage requirements of a compact binary radix search trie for storing and recalling a set of N objects with binary search keys. A compact trie has the property that the number of nodes in the trie is equal to the number of decisions required to isolate each object by its key, plus at most one extra terminal node for each object. Since two disparate keys must differ by at least one bit, only one decision is required to differentiate one key from another. This implies that, for any trie, the number of decisions used to isolate N objects is N-1. Therefore, a best-case trie (for an even N) has no extra terminal nodes, as depicted in Figure C.1, and the number of nodes is

$$N_{bc} = N - 1$$

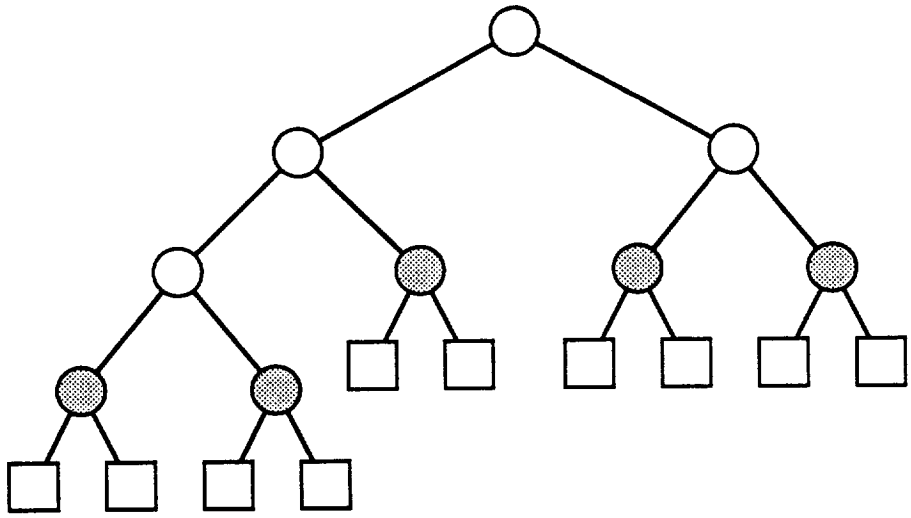

Figure C.1 Best-case compact trie

Likewise, a worst-case trie has an extra terminal node for each object, as Figure C.2 demonstrates, and the number of nodes is

$$N_{wc} = 2N - 1$$

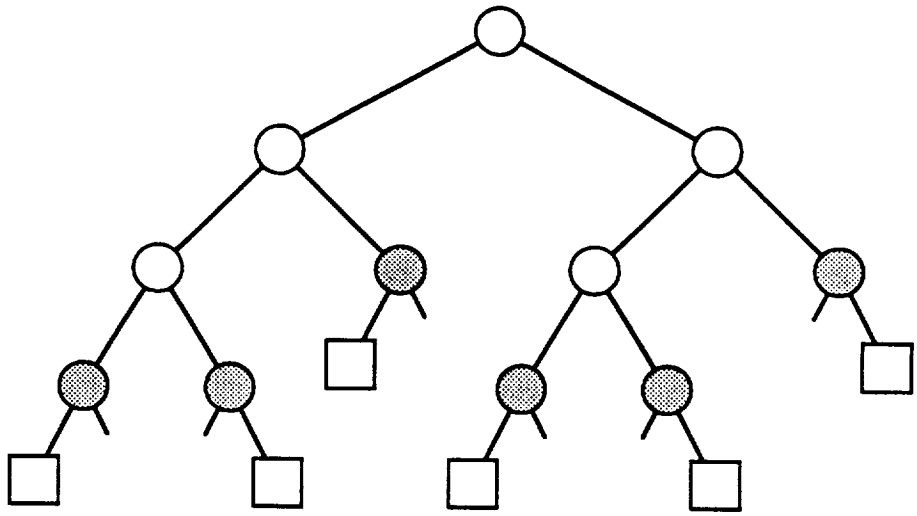

Figure C.2 Worst-case compact trie

It is worth noting that whether a trie for a given object set will be best-case or worst-case does not depend on the overall distribution of keys, as in the full-depth trie; it only depends on the condition that for each object, there is another object whose key differs only by one bit. If this condition is true, then all terminal nodes will be shared, and a best-case compact trie will result.

Appendix D

Glossary for FTS Environment

The definitions of the following acronyms are taken from [25].

FTS	Flight Telerobotic Servicer
IEA	Integrated Equipment Assembly
MSC	Mobile Servicing Center
MT	Mobile Transporter
ORU	Orbital Replacement Unit
RSM	Robot Support Module
SSRMS	Space Station Remote Manipulator System